LLNL-CONF-413143

A Software Quality Engineering Maturity Model

Gregory M Pope, Ellen M Hill

May 19, 2009

Better Software Conference
Las Vegas, NV, United States
June 8, 2009 through June 12, 2009

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Software Quality Engineering Maturity Model

By Gregory M. Pope CSQE and Ellen M. Hill CSQE

Since this paper is about Software Quality Engineering it is appropriate to start by defining how the term will be used in the context of the following discussions. According to Roger Pressman, author of *Software Engineering, A Practitioners' Approach*, everyone feels they understand Software Quality[1]. The definition of software quality varies however. Tom Demarco say's "a product's quality is a function of how much it changes the world for the better"[2], which is pretty subjective and dependent on user satisfaction. Gerald Weinberg's definition is similar, "Quality is value to some person". Steve McConnell points out there are two parts to software quality: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not[3]. Software quality has also been broken down into numerous sub categories (or "ilities"). They are correctness, efficiency, maintainability, portability, scalability, integrity, understandability, reusability, reliability, testability, usability, to name the most common. These quality subcategories are not necessarily independent. For instance, code that lacks correctness (has many defects) will likely be harder to port, scale, or maintain. In fact, correctness could influence every other quality factor.

Most of us have probably had at least some experience with software that lacked quality, especially if we tend to be early adopters of newly released commercial software products or we have tried to code and execute sample problems found in Computer Science text books. In this paper software quality will be treated as the combination of two main characteristics:

1. The software's features meet or exceed the user's expectations

2. The software design minimizes the number and severity of perceivable deviations from expected behavior

The first definition of software quality implies that user's will judge subjectively the value of the software against some preconceived notions of behaviors and feature sets. This definition reminds developers of the importance of understanding the expectations of the user and the norms for the marketplace segment(s) the product competes in. For custom software this could mean fulfilling the requirements of a "one of kind" specification and contract. For software products that sell into mass markets, determining user expectations may be a complex endeavor, requiring extensive marketing research and focus groups. The second definition of software quality implies that the product must also be defect-free enough (have enough correctness) to be reasonably reliable and predictable. Software that has all the features that user's desire will not be perceived as high quality if it crashes every five minutes, likewise, software that is defect free (if that were possible to prove), would not be considered high quality if it was missing the features that users expect and need.

This work performed under the auspices of the U.S. Department of Energy by
Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

Other industries, such as aviation, automotive, and medical, which have existed for years without software, are increasingly using software embedded in their products. The expectations set prior to software being used in these industry products also determine user expectations. It would be unimaginable for a commercial airline pilot to announce on a flight to Denver that the latest version of the flight control software is not compatible with instrument landings above 5,000 feet, and since Denver has fog the plane must land in Phoenix instead. It is common however for modern operating systems to be released without having support for all drivers. Likewise a car manufacturer would be hard pressed to move inventory if their vehicles would not have ABS, climate control, and reverse gear until service pack one is released in six months. We may however see a new operating system on the shelf for sale with features found in previous versions missing. Imagine a medical company offering free radiation treatments while their latest device is being tested and waiting for FDA approval. Operating systems and office software often offer free beta versions. User's expectations can vary for software quality depending on the norms of a particular industry where the software is used.

Engineering can be defined as a discipline which applies technical and scientific knowledge to accurately predict the performance of a design, process, or system in the physical world in order to meet a desired objective or specified criteria. Engineering would be the antithesis of "trial and error". Therefore **Software Quality Engineering** is a discipline that applies technical knowledge and computer science expertise to accurately predict the effectiveness of a software development process to produce a product or service that meets its desired objectives. Software Quality Engineering includes both the process and the product. A practitioner of this activity would be a **Software Quality Engineer**. **Software Quality Assurance** would be the activities that confirm that Software Quality Engineering is taking place on the software engineering process being used, such as an audit against a process standard or an assessment. **Software Quality Control** would assure that Software Quality Engineering is taking place on the product being developed, and would consist of activities such as inspection and testing of the product.

Distinguishing Software Quality Engineering from Software Testing. In the commercial software industry the term Software Quality or Software Quality Engineering or Software Quality Assurance is often used to describe practitioners or groups that primarily engage in system level testing activities. Certainly software testing is part of Software Quality Engineering if it is done using technical and scientific knowledge to predict the outcome of running a test case. The test's outcome is usually expressed as the expected result of running the test case. The expected result is compared to the actual result to determine whether the test passes or fails. Testing has two purposes in theory, to validate that the software meets its requirements and to expose any defects (variations from the requirements or unexpected behavior). All too common in practice however, is a lack of up to date requirements in enough detail to be useful for building test cases. Without adequate requirements, software testing becomes more a defect exposing discipline than a requirements validation discipline. The old adage "you cannot test in quality" stems from the fact that if the software does not do what the customer requires it to do, exposing and removing most of the defects will still not allow the software to do what the customer wants it to do. Testing is a detection activity and commonly used during the software development process (hopefully not just at the end). The creation of good tests (the fewest number of tests that cover the most features and code and find the most defects) is as complex technically as the creation of good code (the fewest lines of code that implement the most desired features and contain the fewest defects). However Software Quality Engineering encompasses many more detection and prevention activities beyond those associated with testing. Software Quality Engineering includes processes that can be used to elicit, analyze and trace requirements, architecture, design, and interface design and optimization, coding standards, inspections and reviews, templates and checklists, audits and assessments, change management, configuration management, static and dynamic code analysis, assurance of compliance to governing standards, determining customer satisfaction levels, analysis of trends and root cause analysis, measurement of the effectiveness of development processes, automation of development processes such as make/build, requirement tracking, defect tracking, design tradeoffs, inspection and review, risk assessment and risk management, as well as all aspects of unit, integration, and system level testing. Many software quality organizations such as the American Society for Quality (ASQ) would also include software project management as another software quality discipline. Certainly having capable software project managers who understand software quality engineering and are experienced developers themselves will increase the likelihood of a project's success.

Who does Software Quality Engineering? Software quality engineering can be accomplished with or without a software quality engineering group or designated person; however success will be more likely with a group dedicated to software quality engineering. Developers are often working to tight deadlines and the main focus of their efforts is to develop software for the end product (or service). When process improvement ideas occur to developers they may not have sufficient time to actually develop the ideas further and implement them. Unfortunately the need for process improvement often becomes obvious after a failure or missed objective, stretching resources even thinner for the remainder of the project. Despite this, if the time needed to make the process improvement is short enough (a few hours); it still may have a chance of getting done by developers. For process improvements that require more than a day to implement, having a person or group dedicated to Software Quality Engineering would increase the chances of the new idea getting developed further, evaluated, and rolled out. Management can also accomplish Software Quality Engineering (SQE), however their primary focus is staffing, budgeting, scheduling, status meetings, work arounds, and risk mitigation. Also, project management may or may not have a background in software development. Understanding that a process needs to be improved is relatively easy. How to improve the process usually requires technical knowledge and a background in computer science. To actually make the improvement usually requires experience in software development.

What about lean environments, agile teams, or smaller companies where there is not the ability to support a software engineering group or engineer? In these environments developers, system testers, and program managers can set aside time to discuss ideas for improving processes. SQE can be a topic of regular scrum sessions or project meetings.

Good ideas may have to wait for the next project if they require a substantial time commitment to implement, or they may have to be accomplished in small steps as time permits. The idea of a best practice forum to share what other small teams are doing may allow leveraging of tools already in use somewhere else in the enterprise. More about Best Practice Forums follows later. The use of an experienced consultant may be another option to jump start a new tool or process. Another option would be a training class to expedite learning a new development process. Another source of good process improvement ideas is to belong to local chapters of software organizations and attend talks. SQE by walking around is another way to gather ideas. Methods of gathering information, some of which may be helpful, others not as helpful, include: attending symposia, listening to speakers, and meandering through the vendor exhibits. Even learning from other industries, being aware of how businesses outside the software industry operate may be a source of ideas.

The success of a software quality engineer or software quality engineering group is greatly enhanced when the members are experienced software developers themselves. There are three main reasons for this:

1. They can interact with developers speaking the same language and gain respect and trust sooner.

2. They have a better appreciation for what the developer's challenges are since they have done similar work themselves.

3. They can use their developer skills to actually implement process improvement tools which are recognized by developers as value added.

Despite these good reasons, all too often those responsible for software quality engineering may not have actual developer experience. When this is the case friction is more likely to develop between developers and those responsible for software quality engineering. Software quality engineers may mistakenly view developers as adverse to quality or process improvement, and software developers may view software quality engineers as roadblocks to getting work done, or worse.

Where to start? The best place to start a software quality engineering program is in the area of testing. There are a couple of reasons for this:

1. Testing is usually not the developer's favorite area of software development. System testing is likely done by a separate group from development. As such, offering to help with system testing usually will not be resisted heavily by developers. What will be resisted heavily by developers is to ask them to spend more time doing design, spend more time analyzing requirements, spend more time inspecting code, more time documenting what they do, or spend more time doing unit testing. Offering to help with system testing does not really require the developers to do more work. Usually system testing groups are willing to accept help, or alternatively developers are willing to accept help with testing.

2. After getting buy in of the developers to help with system testing, make the following offer: If testing shows that there are no defects in the code that the developers are writing, then there is no need to improve the quality of the software development process. If testing shows that there are defects in the code, then the defects will be analyzed to determine how they got into the code and how to detect them earlier or prevent them from getting into the code in the future. Most developers will agree with this logic and tend to be confident that few defects will be discovered.

Of course there will be defects to find in the software under most circumstances. If any kind of decent testing is performed these defects will be exposed. Analysis of the defects may lead the software quality engineer to conclude that upstream prevention and detection techniques should be added to the existing software development process. This can be discussed with developers, who can see the need for the improvements based on what is found during testing, not based on opinion.

An environment where software is being developed to regulatory standards is another way to get a software quality engineering program started. This would be true for government contracts, medical software, safety software, or avionics codes. In this environment it may be difficult for developers to remember all the various compliance regulations that they must adhere to. Software quality engineering can help developers by mastering the myriad of compliance standards and simplifying it to a minimum set of processes, which if followed, make the software developed be in compliance. In some cases government regulations reference other regulations which reference yet other regulations. Sometimes the standards may be ambiguous and even contradictory. Understanding how to apply the standards in a concrete fashion to a particular project may be viewed by developers as helpful and welcomed. The software quality engineering function would be viewed by developers as a protecting function, much like a lawyer advising a client on how not to get sued.

Depending on the particular enterprise, market conditions, and historical factors, getting started with software quality engineering is always easier when the initial tasks involve doing things that are perceived by developers as helpful rather than adding to their burdens. Simply pointing out what developers could improve on without the ability to help solve the problem is of little value, but is not that uncommon.

How to get buy in. Developers are more likely to utilize knowledge and take advice from other developers who are regarded as either their peers or even more experienced in the field. In the software development food chain, developers will be more likely to emulate, aspire and take advice from the highly regarded developers than from a software quality engineer. This is a pretty universal human trait. While the software quality engineer may have previously been a very highly regarded developer, that reputation may not have followed into the new job. Therefore a good strategy to avoid this communication block is to have developers meet and share best practices from other organizations or industry among themselves. A name for this is a best practice forum, which can be a meeting or an e-forum, or both. What is most important is to create a safe environment, similar to a structured inspection session, where it is okay, in fact encouraged, to share openly and honestly what is working well and what is not. Management should be kept out of these forums, and software quality engineering should participate as interested listeners only. The taking of notes, distribution of minutes, updating of forum content, meeting announcements, and other support can be handled by software quality engineering to make it easy for developers to share their information.

Another approach to consider is the idea of pilots or prototypes. This is especially helpful when rolling out new tools or technology. Rather than describe a new tool or technology, the software quality engineer can set up a pilot program using the new tool in a non-interfering way on some small part of the project. For instance a new defect tracking tool could be installed and the data base loaded up with a sampling of defects from whatever is being used to track them currently. The software quality engineer then can ask developers to try the new tool out and evaluate it. This way the developers are actually involved in the decision making process and their perspective can be quite informative. The pilot creates more work for the software quality engineer, but it saves the developer from the time and frustration of getting some new tool up and running in the target environment. If the pilot looks promising, let the developers evangelize the new tool among themselves. When time comes to roll the tool out software quality engineering will participate or do most of the installation tasks. If the pilot is resisted or good reasons are found to not use the tool, only minimal time is lost by developers.

Of course one drawback to the pilot and best practices approach is that software quality engineering may not receive much individual credit for the improvements made. It is therefore important to staff software quality engineering positions with team players who do not mind others sharing or getting credit for improvements. A similar approach was called egoless programming over thirty years ago[4] today this concept is just being applied to software quality engineering as well. The focus is on principles (does the tool or new process actually help or not) and not personalities (somebody with an assertive personality or influence champions the idea). Assuming that an egoless approach is possible in your organization and that you can find some ways to help developers initially, then the challenge becomes how to engage in continuous process improvement for software quality engineering.

Another key to getting started in an organization that has multiple software projects going on simultaneously, possibly managed by different departments, is to utilize a risk based approach to determining the appropriate level of software development process rigor to use on a given software project. The highest level of rigor should be reserved for codes which involve safety directly. If these codes fail people can be killed or seriously injured. A lesser level of rigor could be used on codes whose failure would cause substantial financial loses. Codes that could fail and cause minor financial losses could use a lesser level of rigor. Codes that are prototypes or throw away code would have the least level of rigor. The use of risk consequence and likelihood of failure allows the software project to develop a risk score, and based on the risk score choose an appropriate level of rigor. Levels of rigor could be managed (highest), documented, understood (lowest). A managed level of rigor requires the highest amount of planning, engineering processes, and complement of automated tools. A documented project requires less up front planning but artifacts deliberately and intentionally generated as part of doing the process. Understood requires the least level of rigor, planning, and tools, and only artifacts generated as part of doing the process.[5] What is important is that software quality engineering applies to the enterprise's projects in a graded approach, such that it neither needlessly encumbers projects nor allows important information to escape. A risk based approach also addresses the projects who might otherwise claim that software quality engineering does not apply to them because their project is somehow special (this could include the majority of the enterprise's software projects).

Software Quality Engineering Maturity Model? In order to provide a roadmap for continuous improvement of software quality engineering after getting started, the following maturity model approach is proposed. Most contemporary software developers are comfortable acknowledging the maturity levels of the software development process. What is not talked about much is the maturity level of the software quality engineering process. Software quality engineers are notorious for not taking their own advice, just as many testing tools are notorious for not being fully tested. Perhaps it is time to also look at software quality engineering as a process that has distinct and measurable maturity stages. The staging approach has some distinct advantages:

1. Allows organizations or individuals to identify where they are on the maturity spectrum

2. Provides a roadmap on how to get better at software quality engineering

3. Gives hope to those that have always assumed that the purpose of software quality engineering has been as a watchdog or obstacle that must be tolerated or avoided

To be consistent with the software engineering model, a five stage model is suggested in Table One.

Software Quality Maturity Stages	Characteristics
Stage 1	Whiner or the Software Quality Engineering Know-It-All
Stage 2	Reading & Writing Documents
Stage 3	Measurements
Stage 4	Measurement Based Improvement
Stage 5	Implement Improvements

Table 1. Software Quality Assurance Maturity Stages

Stage One: Whiner or the Software Quality Engineering Know-It-All

In this initial stage of maturity the software quality engineer or group would likely be guilty of at least some of the following:

- Pointing out what is wrong with software development process based on preconceived notions (perhaps derived from a different context)
- Alienating developers. Little empathy for the developer's challenges
- SQE seen as a road block or at best a necessary evil
- SQE not seen as a value added

This stage unfortunately, is the usual starting point for many SQE groups. The group or individual may consist of non-developer types who know just enough about software development to be dangerous. They may be former developers who have had difficulties in the past fitting in on teams. Or they may consist of software quality engineers who have come from a different industry[6] where things are done more formally. They may specialize in Quality Assurance in areas other than software, for instance in mechanical or electrical engineering.

Phrases that may be heard during Stage One are:

Developer	Here comes the SQE, I am going to duck into the men's room.
Manager	I thought adding quality was going to save us money and shorten the schedule
SQE	The developers don't seem to care much about quality
Stakeholder	Have not heard anything about the new release, things must be going well

Whining: Stage one SQE's are eager to point out, ad nauseam, what is wrong with the current software development process. They tend to alienate the development teams with their observations of the obvious and then combine that trait with a lack of ability to offer viable solutions. They do not usually command the respect of the developers, so they must depend on management for empowerment. Management is reluctant to fully support the stage one SQE function because of the complaints that flow back from developers and the delays, which inevitably get blamed on the SQE group. The stage one SQE group see themselves as traffic cops, protectors of cyber space, and they hold developers as mostly slothful malcontents who live to violate rules and release bug filled code. The stage one SQE group see themselves as saviors of the organization, victimized and abandoned by uncaring management that focuses solely on delivery dates and cost cutting.

Negativism: Stage one SQE's often see their job as one of finding fault with the way things are done. The more faults they find, the better they are doing their job. They tend to always see the glass half empty. They know how to make even minor problems sound like the sky will fall and all is lost. Their biggest fear is not being able to find something wrong with a software project, because in their mind this would mean they are not doing their job. Stage one SQE's seemingly have a high probability of winding up on regulatory boards, oversight panels, or audit teams. Their negativity can sometimes be mitigated by large documentation packages (stage two behavior), but the thickness of the documentation is not necessarily a good indicator of the quality of the software. Criticizing developers (or anyone for that matter) without relief will usually alienate the developers to the point that they avoid SQEs altogether. This may explain why so many stage one SQE's are relegated to regulatory boards and oversight panels, nobody wants them on their projects. Finding what is wrong is a relatively easy skill to master, that is why it manifests in stage one. News programs and talk show hosts have long since discovered the ratings power of negative news and soliciting complaints. Few if any Eagle Scouts, relief agency volunteers, or good parents will be featured on the news tonight. While it may be tempting to fall into this popular societal trap, an SQE is not after "ratings" and should always strive to find areas of developer strengths to exploit.

Isolationism: Often the stage one SQE function becomes isolated from the developer group. The SQE group is excluded from important meetings. Developers avoid them, often running the other direction when they are spotted lurking around the developer spaces. The only time these SQE's are sought out is when there is a meeting with the customer or an audit. SQE's can then expound at great length with slick presentations using Power Point slides that show all the quality mechanisms that are in place (and hope no one asks if these procedures are really followed). Having become sufficiently frustrated in stage one, the SQE group is ready to move on to the next level of maturity.

Developer Staffing: One way to move from level one to level two is to get the 'right people on the bus', as Jim Collins discusses in his book "Good to Great'[7]. Collins suggests that for an organization to move from good to great they must have the right mix of people in the organization – with common goals and focus. This can apply to the developer or SQE organization. An example of this concept was a software development project that, due to a reduction in workforce, totally changed their development staff with significant positive results. Prior to the changing of the whole staff, customers were becoming more and more dissatisfied with the errors in the software product when released and the time they (customers) had to spend to debug their applications with each new release. In this case it was critical to restructure the development team with the goal of regaining customer trust for reliable and robust products. The previous team did not understand the need for basic software engineering processes such as configuration and release management and viewed the SQE as a nag about the importance of following accepted software engineering and development practices. Another example was an SQE group that was led and staffed by members who considered themselves far more experienced than the developers they were working with. They continually expected their suggestions to be adopted without explanation or developer buy-in. Eventually this SQE group had to be partially replaced with members who were willing to explain the reason for a process change, get buy-in by demonstrating results using a pilot, and not need to take credit for the improvement. With the right people on the project, both the developer group and SQE group quickly regained customer trust and began delivering a high quality product.

Some of the signs of a project moving from Stage One to Stage Two include:

- Basic processes being followed
- SQE viewed as, at least somewhat, helpful
- Developers don’t run & hide from SQE

Stage Two: Reading and Writing Documents

Some of the traits of Stage Two include:

- Recognize compliance more important than opinion
- Focus on writing standards and processes that implement compliance
- Great documents do not necessarily equate to great software.
- Department of Defense (DoD) historically has emphasized importance of documentation

Documentation: At some point, usually past the mid-point of the software development, somebody will realize that the requirements and design documentation, test cases, and user guide that was promised to the customer has not yet been written. Also, there is no Software Quality Plan or Test Plan. Questions are coming up about what the contract says, or what the company standards require, a typical solution is to use the SQE group to backfill these documentation tasks. Management is happy to do this since it will keep the SQE group out of the developers hair and keep them pinned down to their desks writing instead of continuing to ask management to intervene on their behalf. SQE’s should not only be good at creating documents, but be able to use compliance flow downs and cross-walks to simplify life for developers and the software project.

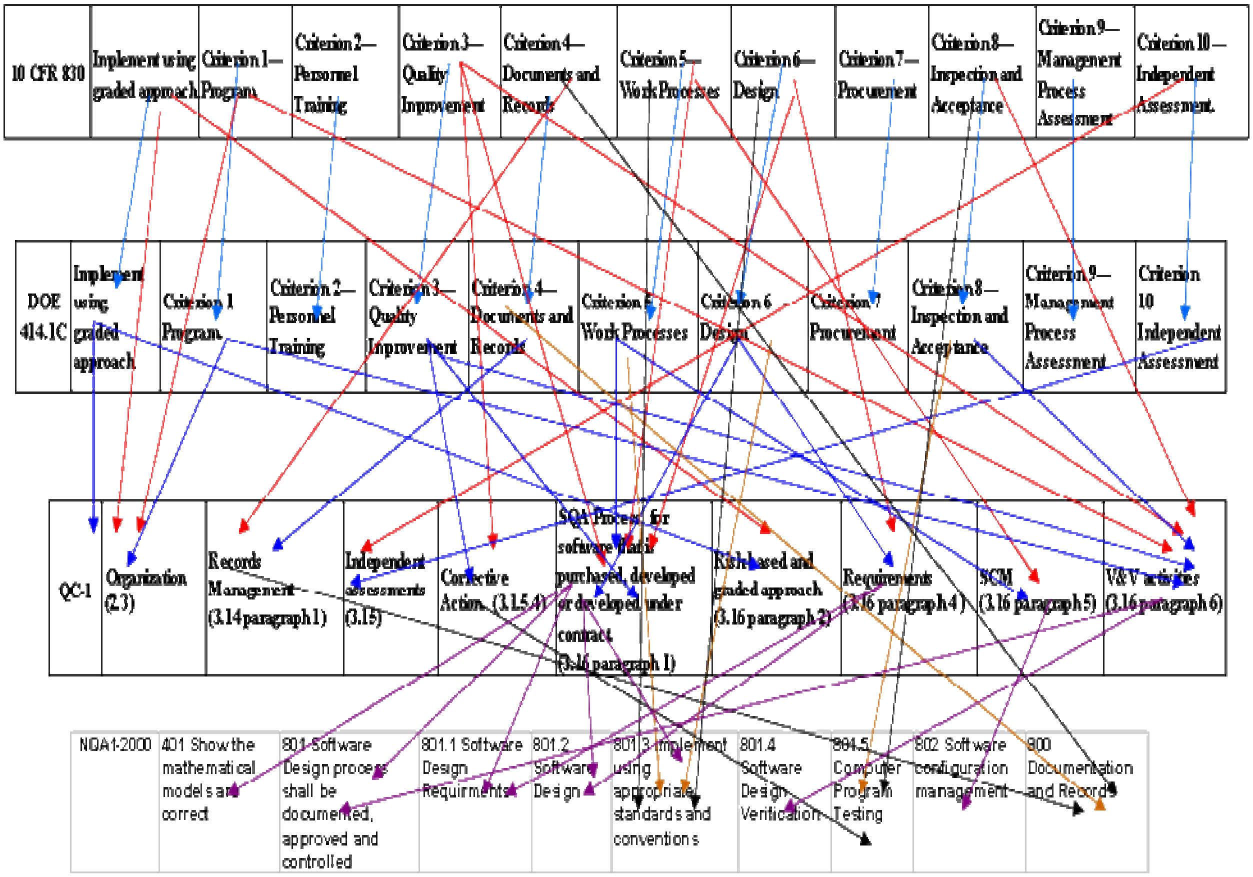
Compliance: A helpful function for SQE to provide for a software development project or enterprise is to understand and communicate the flow down of compliance documents. It would be unreasonable to expect developers to be completely familiar with the details of compliance documents, industry standards, and contractually binding language. Developers have enough on their plates understanding the technical requirements and their development language and environment. SQE can simplify a developer’s life by evaluating the compliance requirements and provide a flow down to show which of these requirements apply to a given software project. This is even more important in regulated industries such as defense, aviation, medical, and government sponsored projects. However, there may well be compliance requirements in commercial enterprises, both internal standards and those imposed by contracts with external partners or customers. What is important for the SQE to do is to determine which set of minimum processes and rigor a development group must follow to be in compliance with these requirements. For instance the SQE function could read and analyze a three hundred page contract and then advise software developers that to be in compliance they need only develop and release code in a configuration management (CM) repository, utilize a defect tracking tool, and develop a set of acceptance tests. This is called flowing down the compliance requirements, a value added task for SQE.

Experienced SQE’s would be able to derive the minimum set of deviations from an enterprise’s standard practices in order to be in compliance, ask for waivers if warranted, and resolve ambiguous requirements. For instance a customer may require that the enterprise deliver bug free code. The SQE would recognize the conundrum of the statement “bug free code” and stipulate that bug free code be defined as a released build passing all customer preapproved acceptance tests. It is clear that the customer wants a good product, but they may not understand the implications of asking for bug free code. The compliance to governing standards will be documented in a Software Quality Assurance plan, usually at the beginning of the document, showing how all the compliance requirements are covered by the rigor of the processes that are used on a given software project.

In creating the Software Quality Assurance plan “gaps” between what is required for compliance and the actual processes being used may be uncovered. The SQE could suggest a waiver with justification or that the process rigor is improved to close the gap. An example of this might be a requirement that test procedures be documented. An SQE may be able to show that the substitution of spreadsheets with test data values and test scripts written in java code using junit, rather than hand written test procedures, will satisfy this requirement. The underlying principle is that tests can be understood before they are run and that they are repeatable.

Crosswalks: Often times the software development project may be under the compliance requirements of multiple standards. For instance one cited by the customer in the contract, and the other an internal enterprise standard. The SQE may develop a cross-walk document to show how the enterprise’s internal standard maps to the external customer’s contract. Using a cross-walk approach the enterprise may be able to show that all the customer’s external compliance requirements will be met if the internal standard is followed. The cross walk may also show gaps that will require waiver with justification or a process rigor improvement. Cross-walks are usually formatted as a table or spreadsheet.

Cross-walks are also useful in heavily regulated industries where multiple agencies may impose multiple standards, which include redundancies and conflicting direction. The SQE function can show with a cross-walk how the sections of different standards are related and where they might conflict. In areas of conflict, one of the standards must be given precedence over the other. For instance, a software safety standard may be cited in the contract, but in another cited standard best commercial practice is required. Prima facie this may seem to be a standards conflict, since developing a code for a safety application requires much more rigor than commercial practice. The SQE could resolve this conflict by assuring the system design relegates the safety code to an isolated platform and power source completely separate from the majority of the code, which is not safety related. Therefore the safety code in the safety subsystem would need to be in compliance with the safety standard, and the remainder of the code in the non-safety related sub system would be built using adequate but less rigorous practices or best commercial practice. Another task for the SQE might be to determine what “best commercial practice” means if that phrase is used in the compliance document, since there is no single best practices set for every software project in every industry. The SQE may cite the enterprise’s internal standard as meeting the requirement of best commercial practice, and of course if the enterprise does not have such a standard, write one. Figure 1 shows an example of cross-walks between multiple standards. Figure 2 shows how the standards flow down into project Software Quality Assurance plans. While all of this sounds simple and makes good common sense, it does require effort and is a task that a stage two SQE should be able to master. The purpose of utilizing an SQE to do flow down and compliance is to relieve the developers from having to remember all the compliance requirements; instead, developers simply need to follow a Software Quality Assurance plan that has been flowed down from various compliance requirements.



10 CFR 830
Implement using graded approach
Criterion 1—Program
Criterion 2—Personnel Training
Criterion 3—Quality Improvement
Criterion 4—Documents and Records
Criterion 5—Work Processes
Criterion 6—Design
Criterion 7—Procurement
Criterion 8—Inspection and Acceptance
Criterion 9—Management Process Assessment
Criterion 10—Independent Assessment.
DOE 414.1C
Implement using graded approach
Criterion 1 Program
Criterion 2—Personnel Training
Criterion 3—Quality Improvement
Criterion 4—Documents and Records
Criterion 5 Work Processes
Criterion 6 Design
Criterion 7 Procurement
Criterion 8—Inspection and Acceptance
Criterion 9—Management Process Assessment
Criterion 10 Independent Assessment
QC-1
Organization (2.3)
Records Management (3.14 paragraph 1)
Independent assessments (3.15)
Corrective Action. (3.15.4)
SQA Process for software that is purchased, developed or developed under contract (3.16 paragraph 1)
Risk based and graded approach (3.16 paragraph 2)
Requirements (3.16 paragraph 4)
SCM (3.16 paragraph 5)
V&V activities (3.16 paragraph 6)
NQA1-2000
401 Show the mathematical models are correct
801 Software Design process shall be documented, approved and controlled
801.1 Software Design Requirments
801.2 Software Design
801.3 implement using appropriate standards and conventions
801.4 Software Design Verification
804.5 Computer Program Testing
802 Software configuration management
800 Documentation and Records

Figure 1. Cross-walks.

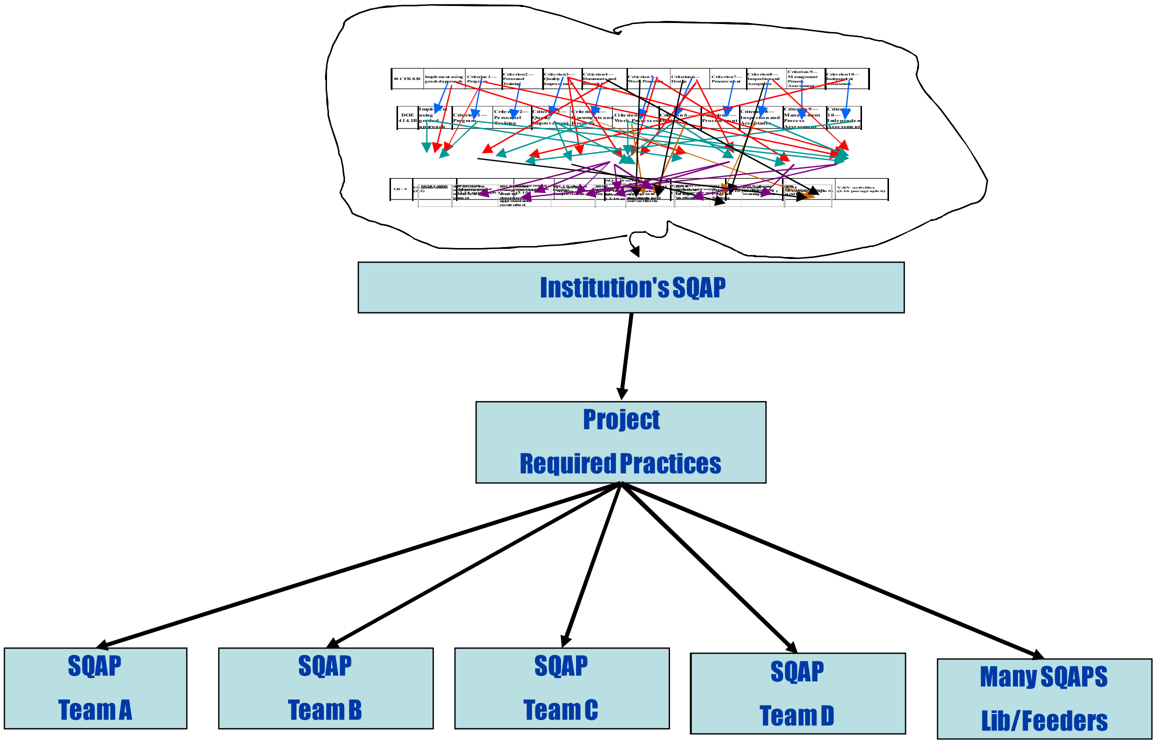


Figure 2. Standards Flow Down into Software Team's Software Quality Assurance Plan

Phrases that may be heard during Stage Two are:

Developer	Glad SQE is off our back, now we can get some work done
Manager	Finally found something useful for SQE to do
SQE	How are we ever going to get all these documents written
Stakeholder	The documentation for the release is thick, should be good

The trouble begins when the Stage Two SQE group realizes that they cannot write the technical requirements down without speaking to the developers. They start asking the developers a lot of questions about the requirements when the developers are trying to meet tight deadlines. Documentation after the fact is of little value, the group soon learns the wisdom of producing artifacts as a result of naturally doing the software development process.

Artifact Capture: During this advanced stage two activity, the SQE group may recommend tools to be used to assist the development team with their code documentation and tracking and analyzing requirements. These tools may include:

- Artifact capture tools: Doxygen to extract formatted documents out of code comments and headers, Source Forge trackers to capture design tradeoffs
- Requirements capture: Source Forge Tracker, Bugzilla, Gira and Round Up to track and analyze requirements
- Configuration Management (CM) tools: CVS, Subversion, PerForce, ClearCase, branching strategies

Some of the signs of a project moving from Stage Two to Stage Three include:

- Checklists exist for processes
- SQE group starts reviewing documents and comparing what the document says to what is really being done (small internal audits or audlets).

Stage Three: Measurements

Stage Three traits may include:

- Realization that developers may not be doing what the documents state, determined by audits.
- Develop a risk based approach to process rigor
- Capture and measure what is really being done
- Calculate the Cost of Quality
- Disaster recovery plans are written
- Inspection and Reviews added
- Static Analysis added
- Defect densities predicted
- Process maturity graded

Compliance: During this stage, the SQE group quickly realizes that the development team is likely not doing what the documents state. They understand the importance of writing short documents that accurately reflect what the project is doing and can be defended to management and stakeholders, as well as pass an audit. The SQE group begins to question what happens if disaster strikes … will all the work be lost, how can it be recovered, who is responsible for such actions? The response to these questions is typically included in a written disaster recovery plan for the project. If an organization has a broader recovery plan the project needs to understand that plan and reference it.

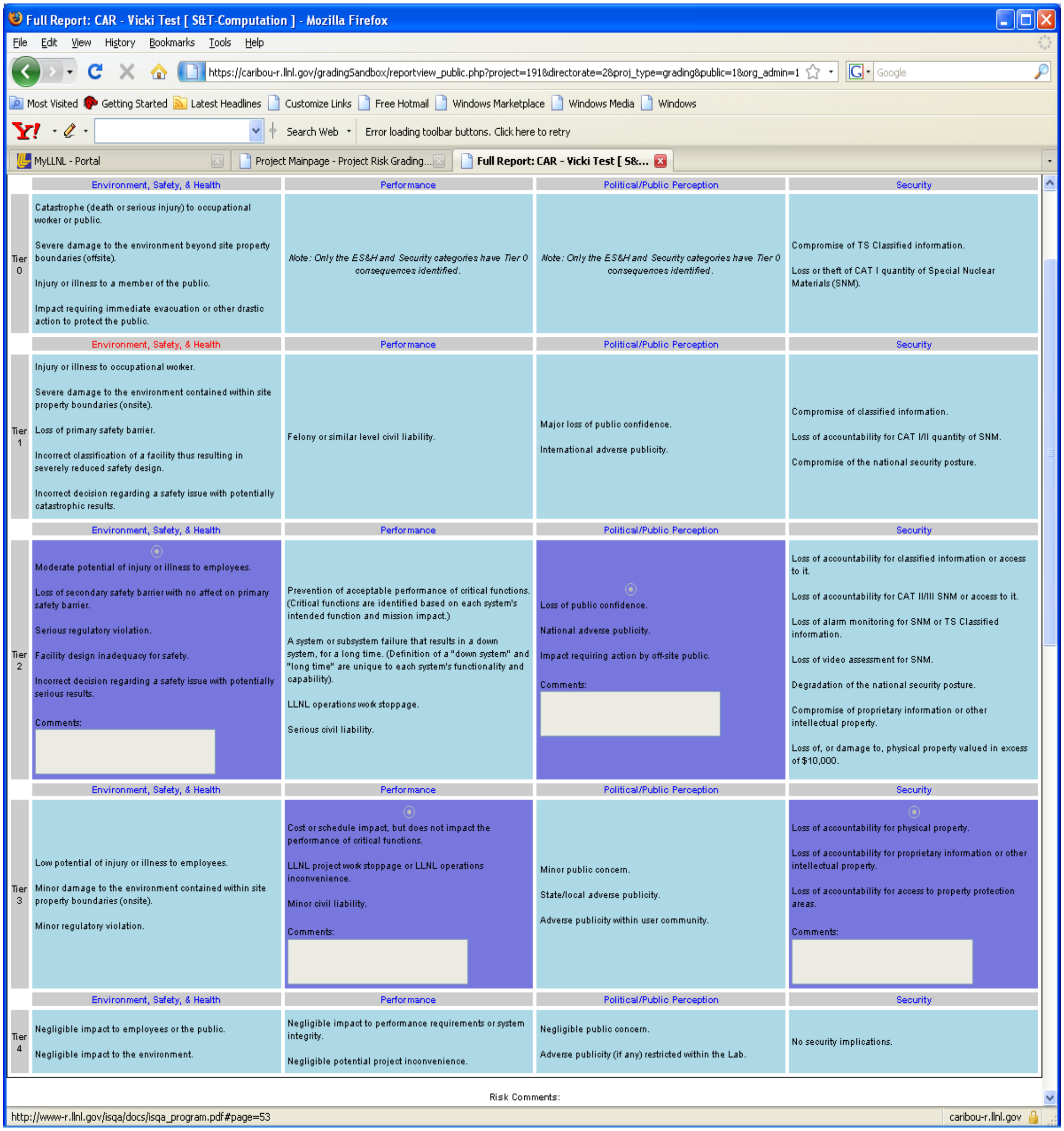
Audlets: One way to gather information about the project processes with minimal developer time is to conduct an 'audlet'. This is an informal 'mini-audit' where the SQE, usually following a checklist, interviews the development team to learn exactly what they do and how they do it. The SQE then writes the documents and has the team review them for accuracy and completeness. By using this approach, both the development team and SQE group have a common understanding of what is being done and what is contained in the documents.

Customer Surveys: A complementary approach to the 'audlet' is customer satisfaction interviews. The SQE group interviews the key customers of a given product. They ask the customers what is working and what is not working with regards to the software product and interactions with the development team. The customer responses will guide the SQE in understanding what needs to be documented and what needs to be measured and improved.

Risk Based Approach: In order to assess the level of rigor needed for any given project, it is important to understand its risk consequence and likelihood of failure. The risk consequence addresses the impact to the Environment, Safety & Health, Performance, Security and Political & Public Perception if the product fails to perform as expected. Table 2a provides a general description for each risk consequence category and Table 2b is an example automated tool.

Risk Consequence Category	**Description**
Environment, Safety & Health	Risks to the operating and external environment, including: toxic release and cleanup. Risks to life and limb. Risks of regulatory liability.
Performance	Risks to meeting program requirements/goals. Risks of system downtime and work stoppage. Risks to the acceptable performance of critical functions, including civil liability. "Critical functions" are those important to the operation of the system or subsystem.
Political & Public Perception	Risks to governmental and public confidence and concerns.
Security	Risks to program, product and material security.

Table 2a. Risk Consequence Categories


Full Report: CAR - Vicki Test [S&T-Computation] - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://caribou-r.llnl.gov/gradingSandbox/reportview_public.php?project=191&directorate=2&proj_type=grading&public=1&org_admin=1
Google
Most Visited Getting Started Latest Headlines Customize Links Free Hotmail Windows Marketplace Windows Media Windows
Search Web Error loading toolbar buttons. Click here to retry
MyLLNL - Portal
Project Mainpage - Project Risk Grading...
Full Report: CAR - Vicki Test [S&...

	Environment, Safety, & Health	Performance	Political/Public Perception	Security
Tier 0	Catastrophe (death or serious injury) to occupational worker or public. Severe damage to the environment beyond site property boundaries (offsite). Injury or illness to a member of the public. Impact requiring immediate evacuation or other drastic action to protect the public.	*Note: Only the ES&H and Security categories have Tier 0 consequences identified.*	*Note: Only the ES&H and Security categories have Tier 0 consequences identified.*	Compromise of TS Classified information. Loss or theft of CAT I quantity of Special Nuclear Materials (SNM).
Tier 1	Injury or illness to occupational worker. Severe damage to the environment contained within site property boundaries (onsite). Loss of primary safety barrier. Incorrect classification of a facility thus resulting in severely reduced safety design. Incorrect decision regarding a safety issue with potentially catastrophic results.	Felony or similar level civil liability.	Major loss of public confidence. International adverse publicity.	Compromise of classified information. Loss of accountability for CAT I/II quantity of SNM. Compromise of the national security posture.
Tier 2	● Moderate potential of injury or illness to employees. Loss of secondary safety barrier with no affect on primary safety barrier. Serious regulatory violation. Facility design inadequacy for safety. Incorrect decision regarding a safety issue with potentially serious results. Comments:	Prevention of acceptable performance of critical functions. (Critical functions are identified based on each system's intended function and mission impact.) A system or subsystem failure that results in a down system, for a long time. (Definition of a "down system" and "long time" are unique to each system's functionality and capability). LLNL operations work stoppage. Serious civil liability.	● Loss of public confidence. National adverse publicity. Impact requiring action by off-site public. Comments:	Loss of accountability for classified information or access to it. Loss of accountability for CAT II/III SNM or access to it. Loss of alarm monitoring for SNM or TS Classified information. Loss of video assessment for SNM. Degradation of the national security posture. Compromise of proprietary information or other intellectual property. Loss of, or damage to, physical property valued in excess of $10,000.
Tier 3	Low potential of injury or illness to employees. Minor damage to the environment contained within site property boundaries (onsite). Minor regulatory violation.	● Cost or schedule impact, but does not impact the performance of critical functions. LLNL project work stoppage or LLNL operations inconvenience. Minor civil liability. Comments:	Minor public concern. State/local adverse publicity. Adverse publicity within user community.	● Loss of accountability for physical property. Loss of accountability for proprietary information or other intellectual property. Loss of accountability for access to property protection areas. Comments:
Tier 4	Negligible impact to employees or the public. Negligible impact to the environment.	Negligible impact to performance requirements or system integrity. Negligible potential project inconvenience.	Negligible public concern. Adverse publicity (if any) restricted within the Lab.	No security implications.

Risk Comments:

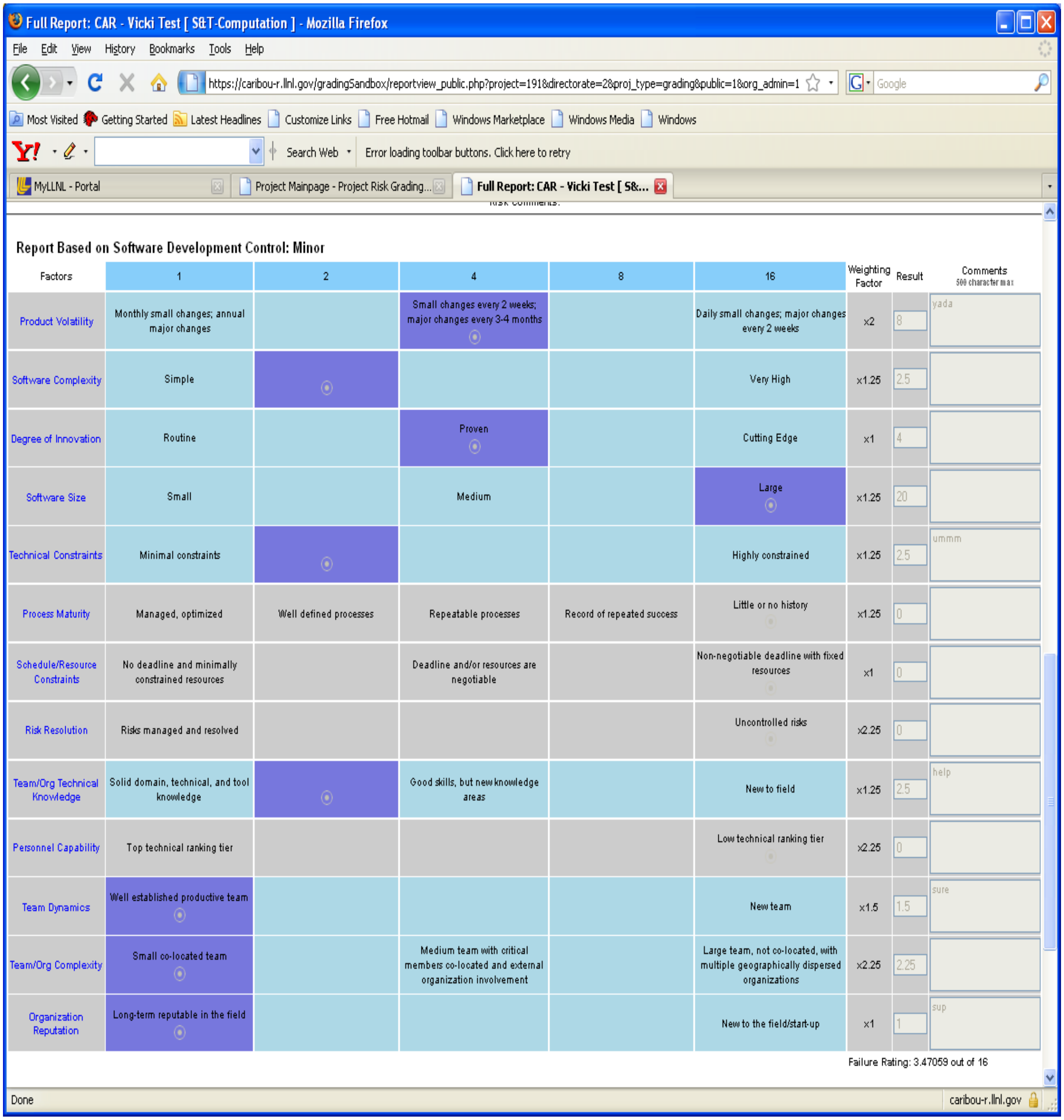
Table 2b. Risk Consequence Tool

Each of these categories has 5 tiers associated with them. Tier 0 is dire consequences and Tier 4 is minimal consequences, with Tiers 1-3 being distributed between them. The SQE group, in consultation with the development team, determines the appropriate risk tier for each category. The highest level tier is used to determine the overall risk level of the product. The professional judgment needed to select the appropriate levels of risk consequence may require the SQE to work with other domain experts. Likewise choosing the proper wording for each risk tier for a particular enterprise may require working with one or more domain experts.

The likelihood of failure calculation takes into consideration many contributing factors of the development environment. Table 3a and 3b below shows some of the more important factors used in determining the likelihood of failure rating. The likelihood table shown was constructed from and weighted according to the COQUALMO risk factors, which trace their pedigree to 40 years of industry studies.[8] Again, the SQE group, in consultation with the development team, determines the appropriate score for each factor. These are then weighted and combined with the risk consequence tier to determine the overall risk level as shown in Table 4a and 4b.

Likelihood of failure contributing factors	Unweighted likelihood of failure score					Weighting	Likelihood of failure scores
	1	2	4	8	16		
Product Volatility	Monthly small changes; annual major changes		Small changes every 2 weeks; major changes every 3-4 months		Daily small changes; major changes every 2 weeks	1.00	
Software Complexity	Simple				Very high	2.25	
Degree of Innovation	Routine		Proven		Cutting edge	1.50	
Software Size	Small		Medium		Large	2.25	
Technical Constraints	Minimal constraints				Highly constrained	1.25	
Process Maturity	Managed, optimized	Well defined processes	Repeatable processes	Record of repeated success	Little or no history	2.25	
Schedule & Resource Constraints	No deadline and minimally constrained resources		Deadline and/or resources are negotiable		Non-negotiable deadline with fixed resources	1.00	
Risk Resolution	Risks managed and resolved				Uncontrolled risks	1.25	
Team/Org Technical Knowledge	Solid domain, technical, and tool knowledge		Good skills, but new knowledge areas		New to field	1.25	
Personnel Capability	Top technical ranking tier				Low technical ranking tier	1.25	
Team Dynamics	Well established productive team				New team	1.00	
Team/Org Complexity	Small collocated team		Medium team with critical members collocated and external organization involvement		Large team, not collocated, with multiple geographically dispersed organizations	1.25	
Organization Reputation	Long-term reputable in the field				New to the field/ start-up	1.00	
Weighted Factor Score Subtotal						18.5	
Weighted Average	Weighted Factor Score Total/Weighting Factor Total (18.5)						

Table 3a. Likelihood of Failure Table


Full Report: CAR - Vicki Test [S&T-Computation] - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://caribou-r.llnl.gov/gradingSandbox/reportview_public.php?project=191&directorate=2&proj_type=grading&public=1&org_admin=1
Google
Most Visited Getting Started Latest Headlines Customize Links Free Hotmail Windows Marketplace Windows Media Windows
Y! Search Web Error loading toolbar buttons. Click here to retry
MyLLNL - Portal
Project Mainpage - Project Risk Grading...
Full Report: CAR - Vicki Test [S&...
Done
caribou-r.llnl.gov

Report Based on Software Development Control: Minor

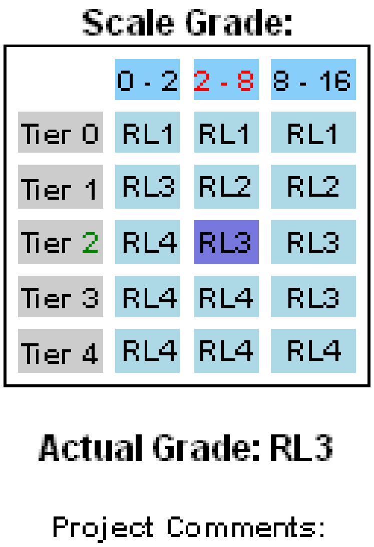
Factors	1	2	4	8	16	Weighting Factor	Result	Comments 500 character max
Product Volatility	Monthly small changes; annual major changes		Small changes every 2 weeks; major changes every 3-4 months ◉		Daily small changes; major changes every 2 weeks	x2	8	yada
Software Complexity	Simple	◉			Very High	x1.25	2.5	
Degree of Innovation	Routine		Proven ◉		Cutting Edge	x1	4	
Software Size	Small		Medium		Large ◉	x1.25	20	
Technical Constraints	Minimal constraints	◉			Highly constrained	x1.25	2.5	ummm
Process Maturity	Managed, optimized	Well defined processes	Repeatable processes	Record of repeated success	Little or no history ○	x1.25	0	
Schedule/Resource Constraints	No deadline and minimally constrained resources		Deadline and/or resources are negotiable		Non-negotiable deadline with fixed resources ○	x1	0	
Risk Resolution	Risks managed and resolved				Uncontrolled risks ○	x2.25	0	
Team/Org Technical Knowledge	Solid domain, technical, and tool knowledge	◉	Good skills, but new knowledge areas		New to field	x1.25	2.5	help
Personnel Capability	Top technical ranking tier				Low technical ranking tier ○	x2.25	0	
Team Dynamics	Well established productive team ◉				New team	x1.5	1.5	sure
Team/Org Complexity	Small co-located team ◉		Medium team with critical members co-located and external organization involvement		Large team, not co-located, with multiple geographically dispersed organizations	x2.25	2.25	
Organization Reputation	Long-term reputable in the field ◉				New to the field/start-up	x1	1	sup

Failure Rating: 3.47059 out of 16

Table 3b. Likelihood of Failure Tool

Consequence of Failure Tiers			
Tier 0	RL1	RL1	RL1
Tier 1	RL3	RL2	RL2
Tier 2	RL4	RL3	RL3
Tier 3	RL5	RL4	RL3
Tier 4	RL5	RL5	RL4
	2	**8**	
	Likelihood of Software Failure Rating		

Table 4a. Risk Level Assessment Grading Table



Scale Grade:

	0 - 2	2 - 8	8 - 16
Tier 0	RL1	RL1	RL1
Tier 1	RL3	RL2	RL2
Tier 2	RL4	RL3	RL3
Tier 3	RL4	RL4	RL3
Tier 4	RL4	RL4	RL4

Actual Grade: RL3

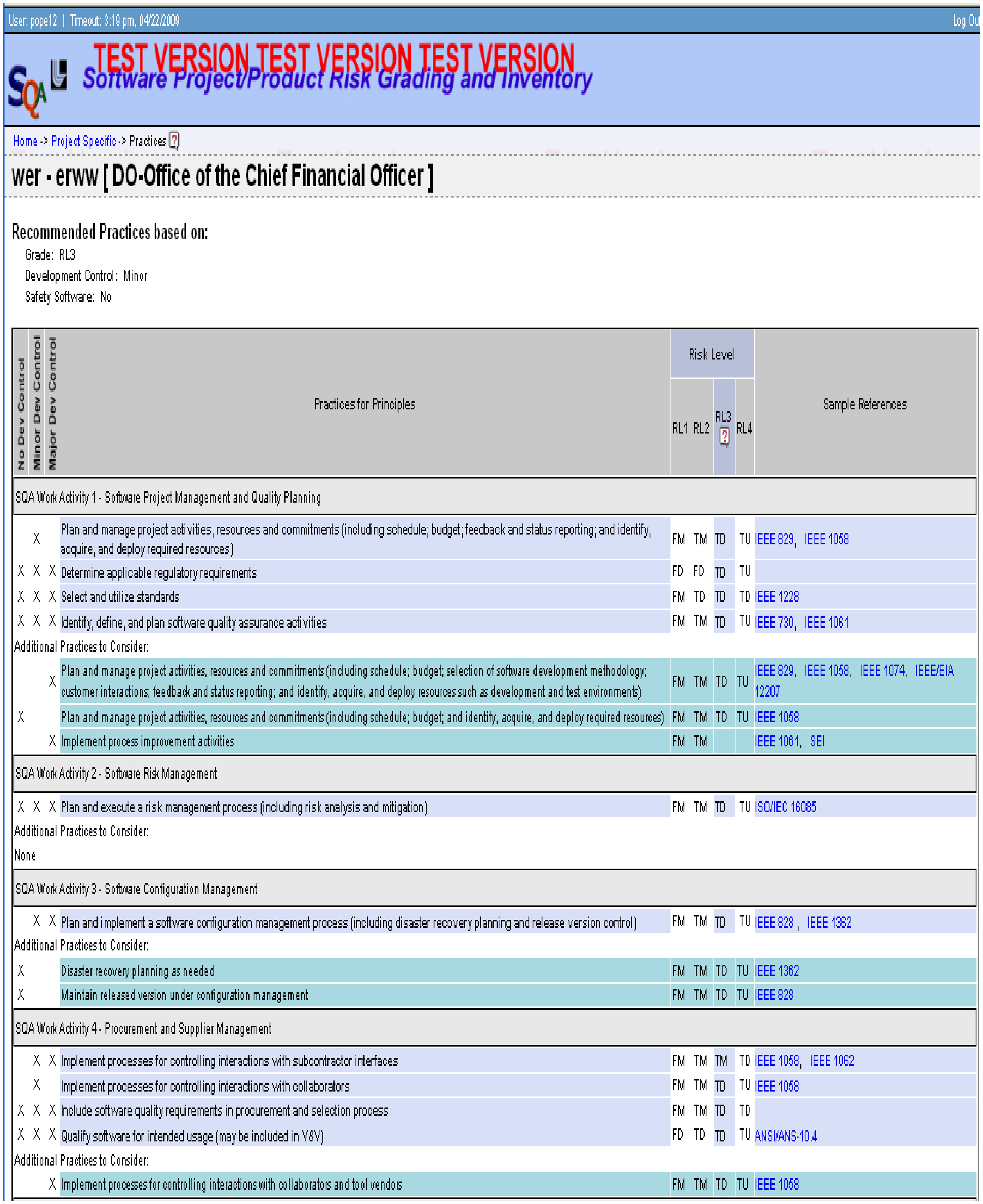
Project Comments:

Table 4b. Risk Level Assessment Grading Tool

This risk analysis results in a better understanding of the overall project risks and level of rigor needed to mitigate those risks. Using the risk assessment tool results in a uniform approach to risk assessment and creates an artifact of the risk assessment process. Risk assessment should be revisited periodically to assure that it is still valid. The risk assessment tool also has tables which delineate the software process activities:

1. Software Project Management and Quality Planning
2. Software Risk Management
3. Software Configuration Management
4. Procurement and Supplier Management
5. Software Requirements Identification and Management
6. Software Design and Implementation
7. Software Safety
8. Verification and Validation
9. Problem Reporting and Corrective Action
10. Training of Personnel in the Design, Development, Use, and Evaluation of Safety Software

The levels of rigor for the software activities span the spectrum from formally managed where software that can cause injury or death requires strict processes while a proof-of-principle code may only require understood practices. Table 4c is an example of the Risk Grading tool's recommended practices for the first four activities.



User: pope12 | Timeout: 3:19 pm, 04/22/2009

Log Ou

TEST VERSION TEST VERSION TEST VERSION

Software Project/Product Risk Grading and Inventory

Home -> Project Specific -> Practices ?

wer - erww [DO-Office of the Chief Financial Officer]

Recommended Practices based on:

Grade: RL3

Development Control: Minor

Safety Software: No

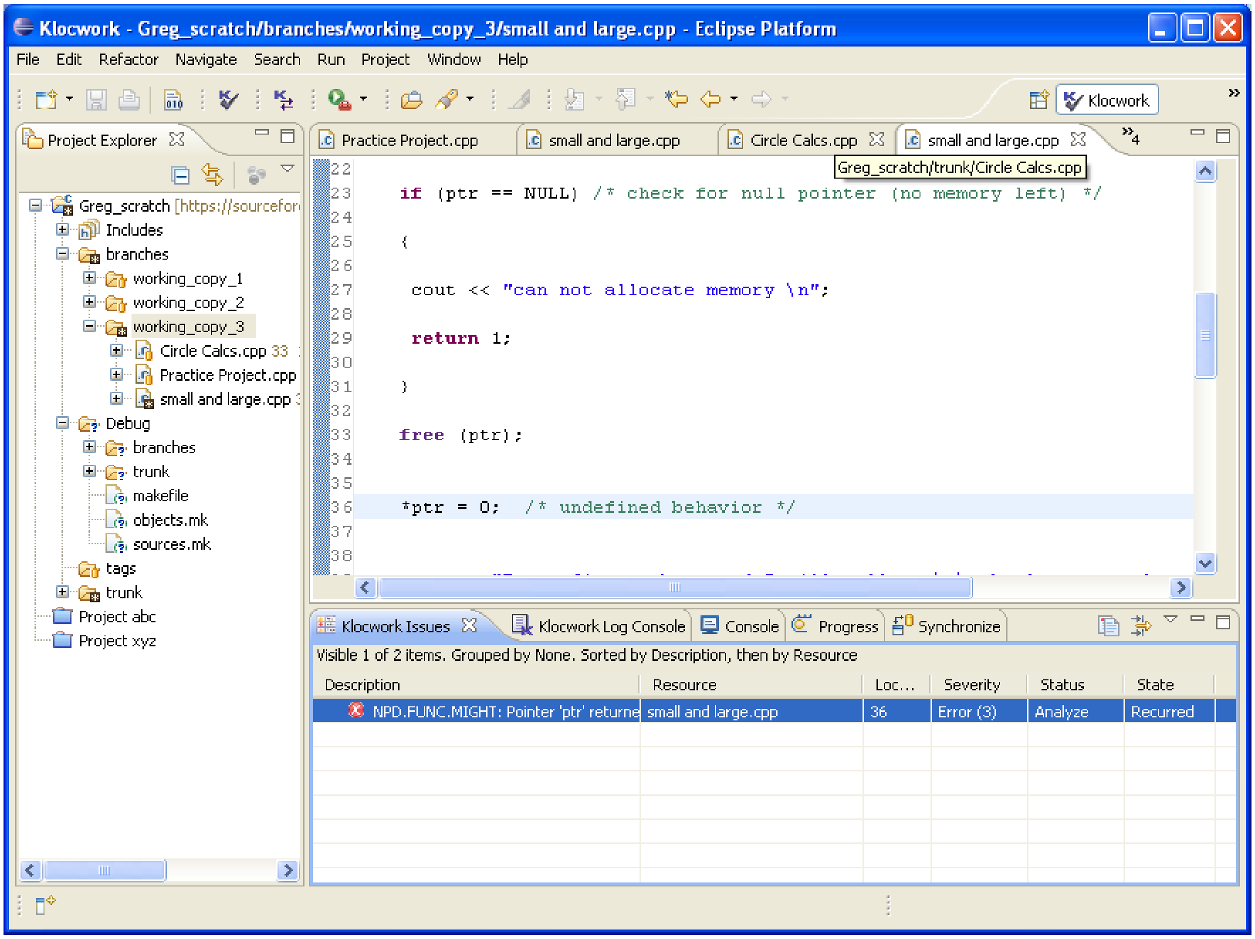
No Dev Control	Minor Dev Control	Major Dev Control	Practices for Principles	Risk Level				Sample References
				RL1	RL2	RL3 ?	RL4	
SQA Work Activity 1 - Software Project Management and Quality Planning								
	X		Plan and manage project activities, resources and commitments (including schedule; budget; feedback and status reporting; and identify, acquire, and deploy required resources)	FM	TM	TD	TU	IEEE 829, IEEE 1058
X	X	X	Determine applicable regulatory requirements	FD	FD	TD	TU	
X	X	X	Select and utilize standards	FM	TD	TD	TD	IEEE 1228
X	X	X	Identify, define, and plan software quality assurance activities	FM	TM	TD	TU	IEEE 730, IEEE 1061
Additional Practices to Consider:								
		X	Plan and manage project activities, resources and commitments (including schedule; budget; selection of software development methodology; customer interactions; feedback and status reporting; and identify, acquire, and deploy resources such as development and test environments)	FM	TM	TD	TU	IEEE 829, IEEE 1058, IEEE 1074, IEEE/EIA 12207
X			Plan and manage project activities, resources and commitments (including schedule; budget; and identify, acquire, and deploy required resources)	FM	TM	TD	TU	IEEE 1058
		X	Implement process improvement activities	FM	TM			IEEE 1061, SEI
SQA Work Activity 2 - Software Risk Management								
X	X	X	Plan and execute a risk management process (including risk analysis and mitigation)	FM	TM	TD	TU	ISO/IEC 16085
Additional Practices to Consider:								
None								
SQA Work Activity 3 - Software Configuration Management								
	X	X	Plan and implement a software configuration management process (including disaster recovery planning and release version control)	FM	TM	TD	TU	IEEE 828 , IEEE 1362
Additional Practices to Consider:								
X			Disaster recovery planning as needed	FM	TM	TD	TU	IEEE 1362
X			Maintain released version under configuration management	FM	TM	TD	TU	IEEE 828
SQA Work Activity 4 - Procurement and Supplier Management								
	X	X	Implement processes for controlling interactions with subcontractor interfaces	FM	TM	TM	TD	IEEE 1058, IEEE 1062
	X		Implement processes for controlling interactions with collaborators	FM	TM	TD	TU	IEEE 1058
X	X	X	Include software quality requirements in procurement and selection process	FM	TM	TD	TD	
X	X	X	Qualify software for intended usage (may be included in V&V)	FD	TD	TD	TU	ANSI/ANS-10.4
Additional Practices to Consider:								
		X	Implement processes for controlling interactions with collaborators and tool vendors	FM	TM	TD	TU	IEEE 1058

Table 4c. Recommended Practices

It should be recognized that doing an initial risk analysis to determine the level of rigor for a software project does not take the place of hazard analysis for safety related codes. The risk analysis would hopefully establish that a software project needs to do hazard analysis because the initial risk assessment has determined that the Consequence of Failure (Severity) and other development risk factors warrant it. Notice also that the process rigor left hand columns (Table 4c) indicate the amount of control an enterprise has over the software project. This can vary from none to major control depending on whether the project is being done in house or by a supplier. Obviously the amount of control over the project is related to the ability to control activities. For each software development activity, the amount of rigor can vary from managed (M) to documented (D) to understood (U). The activities also can be formal (F) or tailored (T). For instance on a RL 1 project, activity 4 "procurement and supplier management", sub activity "qualify software for intended usage" would have to be a formally managed activity, on a RL 4 project the same activity and sub activity could be tailored and understood. After discussions with software safety experts, future versions of this assessment tool will change terminology to not be confused with more formal risk analysis techniques. The terms "consequence of failure" will be changed to "severity", "likelihood of failure" will be changed to "development environment risks" and "risk levels" changed to "quality levels". Allowing the rigor of a software process to be appropriate for the level of risk facilitates the use of a graded approach to an enterprise's software quality engineering process and avoids the challenges and lack of support that is encountered with the "one size fits all" approach.

Inspections and Reviews: There are many ways to find and remove or measure the number of defects in a code. Some of the developer time-intensive processes are code walkthroughs, inspections and desk-checks. In each of these cases, the developers must be prepared to discuss their code sections and review that of their colleagues. Another obstacle to these methods is finding a convenient time and place to hold the sessions. There are some tools that can ease the need for everyone to be available at the same time, such as 'GoToMeeting', CodeStriker and SourceForge Enterprise Edition. These electronic tools allow the team to review code and enter comments at their convenience.

Static Analysis: The SQE group can assist in finding code defects by using static analysis tools on the integrated code builds. A static analysis tool parses the particular language (such as C, C++, C# or Java) code and creates a database that can be used to check every path in the code for 1,600 or so common coding mistakes. These common mistakes could be simple things like not initializing variables, overflowing a string field, dereferencing a null pointer, finding memory leaks or dead code. The ability of the static analyzer tool to check numerous combinations of paths is its big advantage over manual techniques. The reporting capability of these tools also allow for straight forward annotation of the suspected lines of source code and examples of what the tool is complaining about. The tool can also catch errors that are set up pages away from the actual problem line. More sophisticated error types are also detectable, such as referencing a stale memory pointer. At the system level, pre-filtering the results before presenting them to the development team is useful. These tools are capable of finding so many poor coding practices that initial unfiltered reports may overwhelm developers. Pre-filtering by SQEs allows the news to seep out by severity and eliminate a single problem that is replicated in numerous places. SQEs can even become skilled at fixing code or at lease suggesting fixes to developers. An even better use of static analysis is to integrate the capability into the developer IDE as a regular part of their development process. While initially benefiting by finding defects at the system level, earlier detection and removal at the unit level costs less and empowers developers. A sample output is shown in Table 5 below, the tool shown is Klocwork Insight used as a plug in for Eclipse for C++ finding a stale pointer.



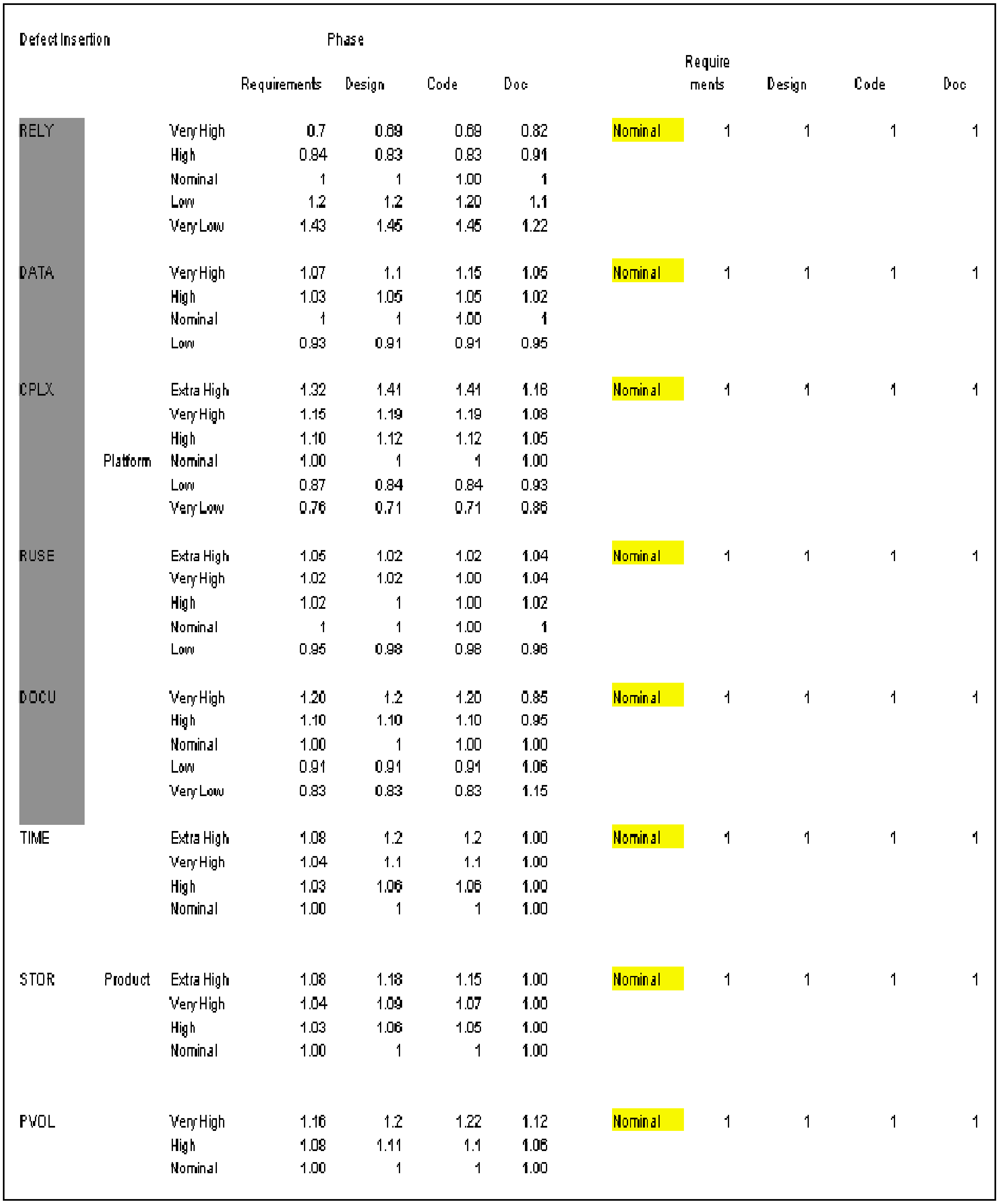
```
    if (ptr == NULL) /* check for null pointer (no memory left) */

    {

     cout << "can not allocate memory \n";

     return 1;

    }

    free (ptr);


    *ptr = 0;   /* undefined behavior */
```

Visible 1 of 2 items. Grouped by None. Sorted by Description, then by Resource

Description	Resource	Loc...	Severity	Status	State
NPD.FUNC.MIGHT: Pointer 'ptr' returne	small and large.cpp	36	Error (3)	Analyze	Recurred

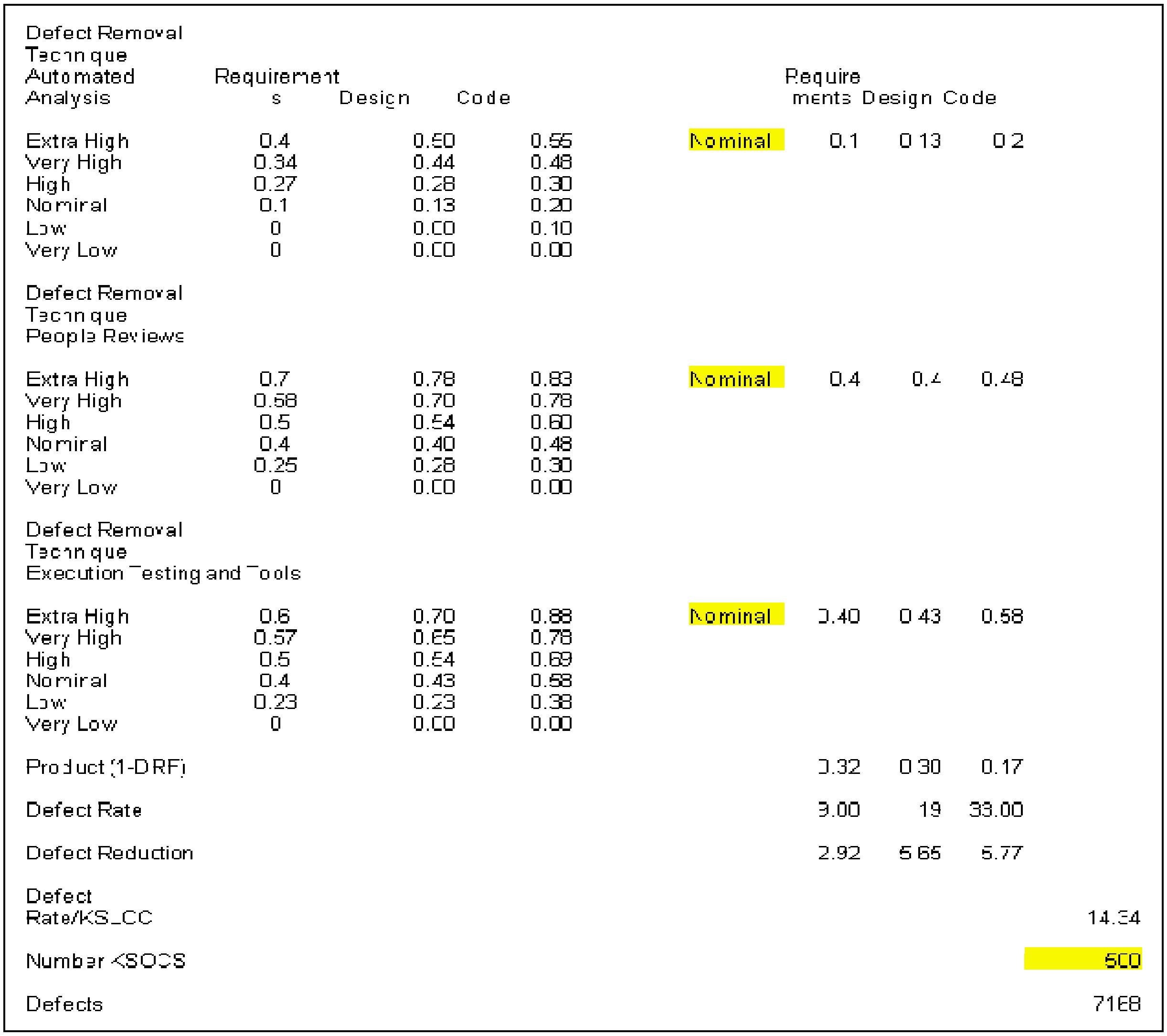
Table 5. Static Analysis Output

There are also several spread-sheet based tools that estimate the defect rate in code, such as Boehm's equivalent CMM level and the COQUALMO estimator. The SQE enters information regarding defect insertion and removal activities into the COQUALMO tool, as shown in Tables 6 and 6a, which are used to calculate the estimated defect rate for the product.   These tools have a couple of benefits, they allow code defect densities to be predicted and then compared against defect densities found using the various detection and removal methods. They also can help the SQE see which activities are allowing the most defects to enter the software development process and which detection activities are not being fully utilized. This can help prioritize future process improvement areas. Information for the tool can come from both developers and SQEs. The analysis can also be used as a “what if“analysis to set goals for defect prevention and detection methods. This tool can be implemented on a spreadsheet.



Defect Insertion				Phase							
			Requirements	Design	Code	Doc		Require ments	Design	Code	Doc
RELY		Very High	0.7	0.69	0.69	0.82	Nominal	1	1	1	1
		High	0.84	0.83	0.83	0.91					
		Nominal	1	1	1.00	1					
		Low	1.2	1.2	1.20	1.1					
		Very Low	1.43	1.45	1.45	1.22					
DATA		Very High	1.07	1.1	1.15	1.05	Nominal	1	1	1	1
		High	1.03	1.05	1.05	1.02					
		Nominal	1	1	1.00	1					
		Low	0.93	0.91	0.91	0.95					
CPLX		Extra High	1.32	1.41	1.41	1.16	Nominal	1	1	1	1
		Very High	1.15	1.19	1.19	1.08					
		High	1.10	1.12	1.12	1.05					
	Platform	Nominal	1.00	1	1	1.00					
		Low	0.87	0.84	0.84	0.93					
		Very Low	0.76	0.71	0.71	0.86					
RUSE		Extra High	1.05	1.02	1.02	1.04	Nominal	1	1	1	1
		Very High	1.02	1.02	1.00	1.04					
		High	1.02	1	1.00	1.02					
		Nominal	1	1	1.00	1					
		Low	0.95	0.98	0.98	0.96					
DOCU		Very High	1.20	1.2	1.20	0.85	Nominal	1	1	1	1
		High	1.10	1.10	1.10	0.95					
		Nominal	1.00	1	1.00	1.00					
		Low	0.91	0.91	0.91	1.06					
		Very Low	0.83	0.83	0.83	1.15					
TIME		Extra High	1.08	1.2	1.2	1.00	Nominal	1	1	1	1
		Very High	1.04	1.1	1.1	1.00					
		High	1.03	1.06	1.06	1.00					
		Nominal	1.00	1	1	1.00					
STOR	Product	Extra High	1.08	1.18	1.15	1.00	Nominal	1	1	1	1
		Very High	1.04	1.09	1.07	1.00					
		High	1.03	1.06	1.05	1.00					
		Nominal	1.00	1	1	1.00					
PVOL		Very High	1.16	1.2	1.22	1.12	Nominal	1	1	1	1
		High	1.08	1.11	1.1	1.06					
		Nominal	1.00	1	1	1.00					

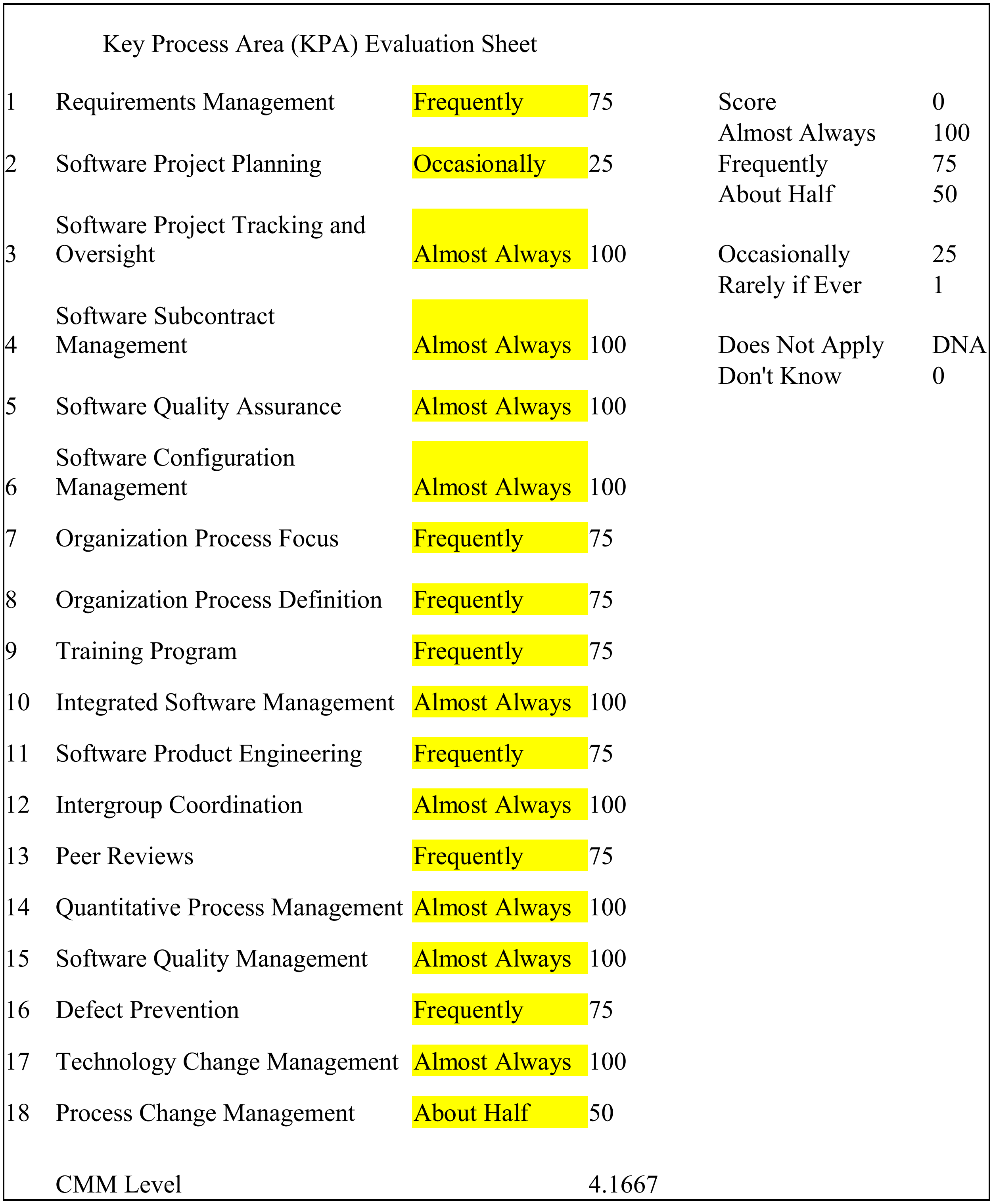
Table 6. COQUALMO Estimator Tool – Defect Insertion Estimates



Defect Removal Technique Automated Analysis	Requirements	Design	Code		Requirements	Design	Code	
Extra High	0.4	0.50	0.55	Nominal	0.1	0.13	0.2	
Very High	0.34	0.44	0.48					
High	0.27	0.28	0.30					
Nominal	0.1	0.13	0.20					
Low	0	0.00	0.10					
Very Low	0	0.00	0.00					
Defect Removal Technique People Reviews								
Extra High	0.7	0.78	0.83	Nominal	0.4	0.4	0.48	
Very High	0.58	0.70	0.78					
High	0.5	0.54	0.60					
Nominal	0.4	0.40	0.48					
Low	0.25	0.28	0.30					
Very Low	0	0.00	0.00					
Defect Removal Technique Execution Testing and Tools								
Extra High	0.6	0.70	0.88	Nominal	0.40	0.43	0.58	
Very High	0.57	0.65	0.78					
High	0.5	0.54	0.69					
Nominal	0.4	0.43	0.58					
Low	0.23	0.23	0.38					
Very Low	0	0.00	0.00					
Product (1-DRF)					0.32	0.30	0.17	
Defect Rate					9.00	19	33.00	
Defect Reduction					2.92	5.65	5.77	
Defect Rate/KSLOC								14.34
Number KSLOCS								500
Defects								7168

Table 6a. COQUALMO Estimator Tool – Defect Removal Estimates

Development Process Maturity Level: For the CMM tool the SQE enters information regarding how often the project addresses key process areas, such as: Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software Subcontract Management, Software Quality Assurance and Management, Software Configuration Management, Organization Process Focus and Definition, Training Program, Integrated Software Management, Software Product Engineering, Intergroup Communications, Peer Reviews, Quantitative Process Management, Defect Prevention, Technology and Process Change Management. These values are then combined to give an estimate of the project's CMM level. A sample is shown below in Table 6b.



Key Process Area (KPA) Evaluation Sheet

1	Requirements Management	Frequently	75	Score	0
				Almost Always	100
2	Software Project Planning	Occasionally	25	Frequently	75
				About Half	50
3	Software Project Tracking and Oversight	Almost Always	100	Occasionally	25
				Rarely if Ever	1
4	Software Subcontract Management	Almost Always	100	Does Not Apply	DNA
				Don't Know	0
5	Software Quality Assurance	Almost Always	100		
6	Software Configuration Management	Almost Always	100		
7	Organization Process Focus	Frequently	75		
8	Organization Process Definition	Frequently	75		
9	Training Program	Frequently	75		
10	Integrated Software Management	Almost Always	100		
11	Software Product Engineering	Frequently	75		
12	Intergroup Coordination	Almost Always	100		
13	Peer Reviews	Frequently	75		
14	Quantitative Process Management	Almost Always	100		
15	Software Quality Management	Almost Always	100		
16	Defect Prevention	Frequently	75		
17	Technology Change Management	Almost Always	100		
18	Process Change Management	About Half	50		
	CMM Level		4.1667		

Table 6b. Equivalent CMM Sample

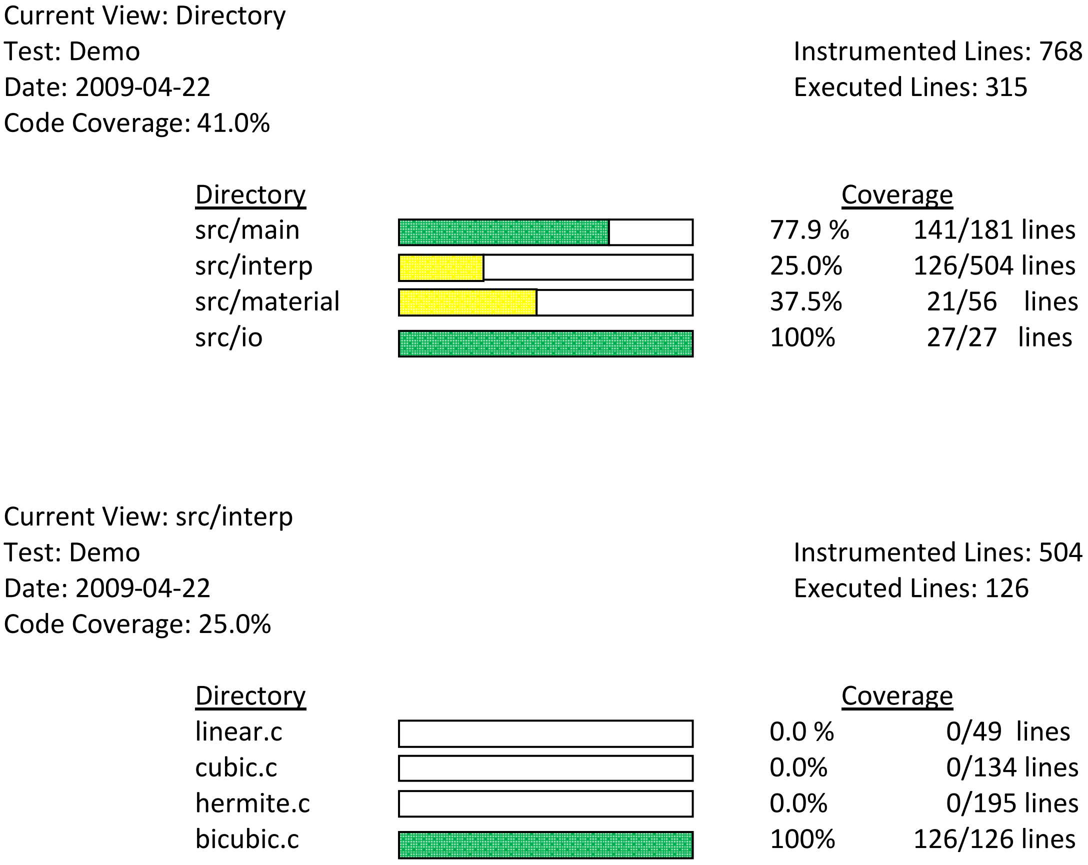
These estimates provide the SQE group and development team a guideline as to how effective their processes are and where improvements may lead to significant decreases in defects. While not as accurate as a formal CMM assessment by a certified assessor the Equivalent CMM tool can give an enterprise a rough idea of what their process maturity level is. The output of this tool can also be used as a cross check of the COQUALMO tool. The defect rate ranges for each CMM level have been collected from industry and are available for comparison to the defect range estimated by the COQUALMO tool. The estimates can also be compared against tables of defect rates by industry compiled by Donald Reifer as shown in Table 6c.

Application Domain	**Number Projects**	**Error Range (Errors/KESLOC)[1]**	**Normative Error Rate (Errors/KESLOC)**	**Notes**
Automation	55	2 to 8	5	Factory automation
Banking	30	3 to 10	6	Loan processing, ATM
Command & Control	45	0.5 to 5	1	Command centers
Data Processing	35	2 to 14	8	DB-intensive systems
Environment/Tools	75	5 to 12	8	CASE, compilers, etc.
Military –All	125	0.2 to 3	< 1.0	See subcategories
§ Airborne	40	0.2 to 1.3	0.5	Embedded sensors
§ Ground	52	0.5 to 4	0.8	Combat center
§ Missile	15	0.3 to 1.5	0.5	GNC system
§ Space	18	0.2 to 0.8	0.4	Attitude control system
Scientific	35	0.9 to 5	2	Seismic processing
Telecommunications	50	3 to 12	6	Digital switches
Test	35	3 to 15	7	Test equipment, devices
Trainers/Simulations	25	2 to 11	6	Virtual reality simulator
Web Business	65	4 to 18	11	Client/server sites
Other	25	2 to 15	7	All others

Table 6c. Defect Rates by Industry

Code Coverage (Dynamic Analysis): Once all the defect rate estimates are done, it is important to then measure the amount of code that the test cases actually execute. If the test cases are missing large portions of the code when run, defects in those areas will go undetected. In addition, it is not uncommon for legacy projects to accumulate large inventories of tests and to assume that test coverage is good based on quantity of tests. It could be that many of these tests are simply redundant (test the same areas of code) and can be eliminated. Most compilers have options that can be turned on to generate the data needed and tools to view the code that was executed during testing. The sample below is derived from the GCC compiler option using the GCOV tool. It shows the overall coverage for the code graphically and numerically and allows the user to drill down within each sub-directory. Using this information, the SQE can develop test cases that execute the most important parts of the code. In this example, the SQE quickly sees that the interp sub-directory has the lowest coverage. By drilling down into that sub-directory the reason for the low coverage immediately becomes clear – only one interpolation method has been executed. Ideally, the SQE would then create new test cases to execute each of the remaining methods and re-run their coverage tool.

Sample GCOV Code Coverage Report



```
Current View: Directory
Test: Demo                                  Instrumented Lines: 768
Date: 2009-04-22                            Executed Lines: 315
Code Coverage: 41.0%

        Directory                                  Coverage
        src/main                          77.9 %      141/181 lines
        src/interp                        25.0%       126/504 lines
        src/material                      37.5%        21/56   lines
        src/io                            100%         27/27   lines


Current View: src/interp
Test: Demo                                  Instrumented Lines: 504
Date: 2009-04-22                            Executed Lines: 126
Code Coverage: 25.0%

        Directory                                  Coverage
        linear.c                          0.0 %          0/49  lines
        cubic.c                           0.0%           0/134 lines
        hermite.c                         0.0%           0/195 lines
        bicubic.c                         100%        126/126 lines
```

Code coverage tools usually allow the granularity of the coverage to measure individual source statements or functions. Code coverage tools do alter the way the software being measured is executed, and can add considerable execution time. Measuring statement coverage will add more execution time than measuring function coverage. Coverage tools do have some drawbacks besides slowing execution time, they do not test all data states, and the tests used to measure the code coverage may not execute the software under test in the same ways that a user might.

Two other useful dynamic analyzers are profilers, such as GPROF, and leak detectors such as Purify. Profilers indicate the amount of time that the software spends in different parts of the code. They can help find bottlenecks and areas of code that are candidates for optimization techniques, such as an algorithm redesign (e.g. bubble sort changed to binary sort) or use of a faster language (use compiler optimization or write the function in C or assembler). Profilers can also be used to instrument code that is at customer locations to better understand which functions customers use most frequently. Leak detectors are a nice complement to static analyzers. Dynamic analyzers can find situations where main memory or disk space resources are depleted, race conditions and problems unique to multi-threading or parallel implementations exist, or interactions with other system resources that cause problems which only occur during execution.

Return on Investment: Another measurement that management will want to understand before embarking on a software quality expedition is 'how much does this cost?' and 'Is it worth the expense?' True, the development of a quality product is not free. There are the obvious costs of developer time, hardware on which to work, development environments, test equipment, etc. Other costs, such as the impact of customer dissatisfaction, can be more difficult to estimate.

There are four major categories to consider in estimating the cost of quality[9]:

- Prevention – quality planning, formal technical reviews, test equipment, training
- Appraisal – peer reviews, testing, equipment calibration & maintenance
- Internal Failure – defects found before release such as debugging, repair, rebuild
- External Failure – defects found after release such as complaint resolution, product return, help desk support, warranty work

If an organization has effective processes for prevention, appraisal and internal failure activities it may expect that their external failure costs will be minimized. Barry Boehm and others[10] have collected data that shows that the relative cost of correcting errors increases exponentially as the development proceeds from requirements through implementation. There is much debate over what the appropriate percentages for each category should be for any organization to estimate its cost of quality. One study[11] found that initial estimates needed to be revised as it took about 6 months to understand how the prevention and appraisal activities affected the internal and external failures. Typically more time is spent on prevention and appraisal costs with the hope of minimizing the cost of internal and external failures. As noted above, it is sometimes difficult to determine the minimal cost of quality due to uncertainty in the external failure costs. Table 7, below, is a diagram of the cost of quality balance[12]. In this diagram the total cost of quality is the sum of all four categories. The appraisal and prevention costs will increase as the code becomes better, which implies more errors are being found in these phases. The internal and external failure costs decrease as the code becomes better because errors are found in the earlier phases.

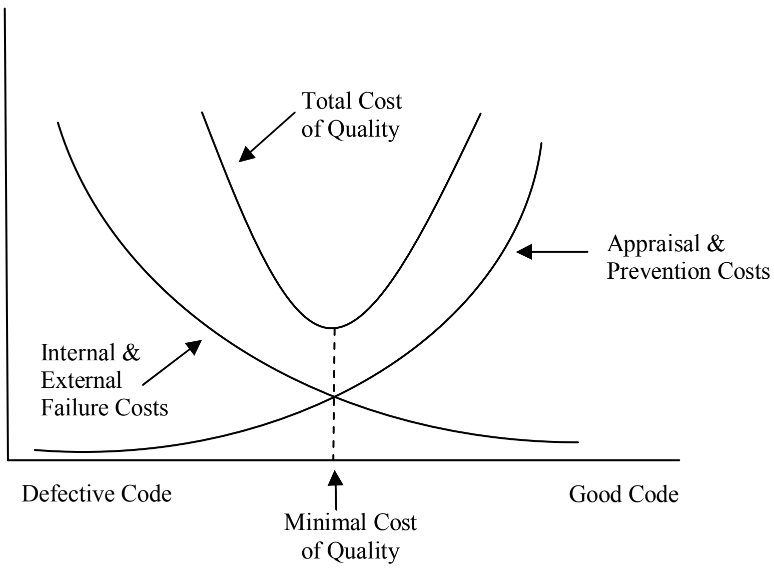
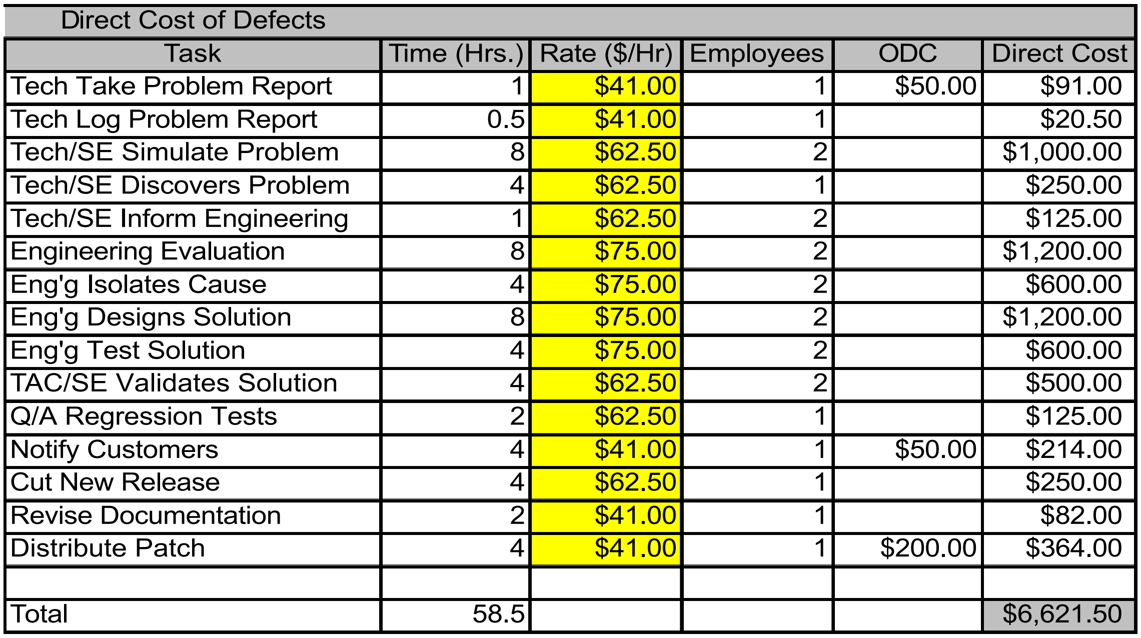


Table 7. Cost of Quality Balance

Example Return on Investment of Process Improvements - Cost of Defects : Understanding the general concept of the cost of quality is one thing, but how does a Software Quality Engineer go about applying this theory in order to decide if a quality improvement is worth the cost? One of the important metrics to determine for justifying a software process improvement is the direct cost of a software defect. Direct costs are those which can be directly measured. Indirect costs include lost revenue, litigation, higher insurance costs, higher costs of sales, longer sales gestation period, etc. and will not be considered as part of this analysis. In order to approximate direct cost of a defect for your project or enterprise a list of tasks was developed that represent the various actions that must be accomplished to remedy a defect found after delivery. These tasks are shown in Table 8. Next, average hours to accomplish the tasks were estimated. This is multiplied by the direct labor cost including 30% overhead for benefits. In some cases more than a single employee is involved with the task. The ODC column includes non-labor costs such as telephone charges, media costs, etc. The chart below (Table 8) indicates that the average direct cost of a defect for our example project is $6,621.50. Of course some defects may cost much more than this, especially if foreign travel is involved. For the sake of this analysis the average cost of a defect will be used.



Direct Cost of Defects					
Task	Time (Hrs.)	Rate ($/Hr)	Employees	ODC	Direct Cost
Tech Take Problem Report	1	$41.00	1	$50.00	$91.00
Tech Log Problem Report	0.5	$41.00	1		$20.50
Tech/SE Simulate Problem	8	$62.50	2		$1,000.00
Tech/SE Discovers Problem	4	$62.50	1		$250.00
Tech/SE Inform Engineering	1	$62.50	2		$125.00
Engineering Evaluation	8	$75.00	2		$1,200.00
Eng'g Isolates Cause	4	$75.00	2		$600.00
Eng'g Designs Solution	8	$75.00	2		$1,200.00
Eng'g Test Solution	4	$75.00	2		$600.00
TAC/SE Validates Solution	4	$62.50	2		$500.00
Q/A Regression Tests	2	$62.50	1		$125.00
Notify Customers	4	$41.00	1	$50.00	$214.00
Cut New Release	4	$62.50	1		$250.00
Revise Documentation	2	$41.00	1		$82.00
Distribute Patch	4	$41.00	1	$200.00	$364.00
Total	58.5				$6,621.50

Table 8. Direct Cost of Defects

Cost of Process Improvement: The next item to be considered is the initial cost of improving the software development process as shown in Table 9. The initial cost is a one-time charge for improving the software development process. There also may be a recurring cost of doing the new process and this will also be considered. There may be a number of suggestions for improving the project's software development process, but for the example in this analysis adding inspections will be used. Again all the tasks are delineated, with direct labor cost and overhead multiplied by hours. In some cases the costs are quite high because of the number employees that need to be trained. According to the assumptions, the initial cost of adding inspections will be $70,538.

Cost of Adding Inspections Process					
Task	Time (Hrs.)	Rate ($/Hr)	Employees	ODC	Direct Cost
Create Preliminary Concept	4	$185.00	1		$740.00
Create Management Presentation	16	$185.00	1		$2,960.00
Obtain Management Buy In	2	$185.00	1		$370.00
Select Tailoring Team	2	$185.00	2		$740.00
Select Process Baseline	7	$75.00	2		$1,050.00
Tailor Process	16	$75.00	6		$7,200.00
Create Process Description	40	$75.00	2		$6,000.00
Create Work Product Templates	16	$75.00	2		$2,400.00
Create Work Product Checklists	16	$75.00	2		$2,400.00
Create Training Course	40	$75.00	1		$3,000.00
Create/Modify Tracking Tool	40	$75.00	1		$3,000.00
Select Metrics	20	$75.00	6		$9,000.00
Train Pilot Team	4	$75.00	7		$2,100.00
Pilot Process	4	$62.50	5		$1,250.00
Evaluate Pilot	4	$75.00	6		$1,800.00
Improve Process based on Pilot	4	$75.00	2		$600.00
Schedule Training	4	$41.00	1		$164.00
Conduct Training	4	$68.75	80		$22,000.00
Roll Out/Evangelize Process	4	$75.00	6		$1,800.00
Collect metrics	4	$41.00	1		$164.00
Re-evaluate and Improve Process	4	$75.00	6		$1,800.00
Total	255				$70,538.00

Table 9. Cost of Adding Inspections Process

Recurring Cost of Inspections					
Task	Time (Hrs.)	Rate ($/Hr)	Employees	ODC	Direct Cost
Author Selects and Notify Inspectors	1	$75.00	1		$75.00
Author Selects Moderator/Scribe	,5	$75.00	1		
Reserve Conference Room	0.5	$41.00	1		$20.50
Author Distributes Materials	1	$75.00	1		$75.00
Kick Off Meeting	1	$68.50	5		$342.50
Inspectors Inspect Work Product	2	$68.50	5		$685.00
Inspection Meeting	1.5	$68.50	5		$513.75
Defects Logged Into Tracking System	1	$62.50	1		$62.50
Author Corrects Defects	8	$75.00	2		$1,200.00
Inspectors Verify Fixes	1	$68.50	5		$342.50
Defects Closed	1	$62.50	2		$125.00
Metrics Updated	1	$62.50	1		$62.50
Follow Up	1	$68.50	2		$137.00
Total	20				$3,641.25

Table 10. Recurring Cost of Inspections

Recurring Cost of Process Improvement: In addition to the initial cost of $70,538, there will be a recurring cost of $3,641.25 for every inspection that is held as shown in Table 10.

Return on Investment of Process Improvement: This last chart (Table 11) utilizes the costs associated with the cost of defects and the initial and recurring cost of inspections to determine the return on the investment. To determine the return, additional information is needed. The average number of defects found per inspection is 7.5. This number comes from an inspection process used by a major Silicon Valley software company for a one year period by 450 employees. There are industry studies that would indicate that 7.5 is a conservative number and many more defects may actually be found. Therefore the break-even point is 1.53 inspections, meaning that halfway through the second inspection the cost savings of finding and removing defects would cover the cost of adding inspections. Every time an inspection is held a $46,020 savings is realized. Assuming that 4 inspections per month are held, the break-even would occur in week 2 of the first month and the return on the investment over a year would be approximately 10 times. The return on investment calculation would indicate implementing the inspection process improvement would be a good investment in terms that a Chief Financial Officer would understand.

Direct Cost of Defects	$6,621.50
Direct Cost of Adding Inspection Process Step	$70,538.00
Cost of Each Inspection	$3,641.25
Average Number of Defects Found per Inspection (Industry Actual)	7.5
Break Even (in number of inspections)	1.53
Inspections per month	4
Break Even (in months)	0.38
Cost of Inspections for a year (including start up)	$245,318.00
Savings in removed defects for a year	$2,383,740.00
Return on investment for a year	9.72

Table 11. Return on Investment of Process Improvement

Phrases that may be heard during Stage Three are:

Developer	Tell me what you want to measure and I'll be sure to do it
Manager	I need to keep the project on track
SQE	We don't care who made the errors, just want to minimize them
Stakeholder	What do I get out of these measurements

Some of the signs of a project moving from Stage Three to Stage Four include:

- Recognize need for templates, may start developing them from checklists
- Defect rates are being calculated and reviewed
- Code coverage results indicate new tests needed

Stage Four: Measurement Based Improvement

Now that so many measurements have been made on the code and processes, this stage focuses on what improvements may be made based on the measurements. The questions asked at this stage tend to be related to how to improve, how to use the information from the earlier stages to make improvements that have a positive impact.

Risk Grading Tool: The use of a tool to do the risk-based grading of the software product enables documentation of the decisions made and reasons why, which may become important in audits and future use of the product. Either the SQE group or the development team or a combination can enter the information into the tool and the rest of the team members, including management, have the opportunity to review, discuss and concur on the decisions. This leads to a common understanding of the project risks and mitigation strategies being used. The Risk Assessment tool can be used in stage four to push further down into the software "ecosystem". Not only are primary codes risk assessed, but the supporting libraries, feeder codes, procured codes, and open source codes are individually assessed. It would be important to identify cases where a Risk Level 4 code was providing important data to a Risk Level 1 code. There are cases however where it may be feasible to have a lesser risk level code feed a higher risk level code. For instance, a risk level 3 code uses a risk level 4 code's display software. Since the display software does not alter the computations in the risk level 3 code, it may be possible to devise a mitigation strategy, by requiring the risk level 4 code to pass a set of tests that are built from the risk level 3 code's outputs. This additional testing on the risk level 4 code may then allow it to be used to support a risk level 3 code because the risk of using it has been mitigated with additional testing.

Feature Coverage: For one code team, statement coverage did not provide all of the information they were looking for so they worked with their SQE to implement feature coverage. The results of which led the developers to champion feature coverage and extended it to count the number of times a feature was executed and by which test case.

Feature coverage was added by modification of the source code rather than by adding a tool. Feature coverage results led to the creation of a feature based test matrix that is used to selectively test any new capabilities that are added. In other words if a developer wants to modify feature A, he or she could determine which regression tests to run for that feature, in addition to whatever unit tests the developer has already run.

Levels of Test Rigor: Test code coverage along with test timing data results are quite helpful in determining which test cases are most useful for developing different levels of system tests, such as smoke, nightly and release test suites. The smoke tests might be used for regression testing prior to developer check in. They run fast, no more than 30 minutes, so they will tend to get used. They would consist of the quickest running tests that achieve the most code coverage. Nightly tests could take three or four hours, and achieve great regression test coverage. Release tests may take a couple of days, but are not frequently run, usually just prior to a scheduled release.

Risk Management: In order to understand which improvements will be most beneficial to a project, it is important to know what concern or risk they are intended to address. This implies that the project has a risk management plan. The SQE and project team work together to identify, assess, analyze, track, review and ultimately retire risks. When considering implementation of a process improvement activity, the team will review their risks and analyze the impact of the activity on the overall success of the project. An effective risk management plan is one that is dynamic and constantly reviewed. Risk Management plans are frequently written once early on and not consulted for the remainder of the project. To avoid this, consider putting the risks in a tracker tool and update them as part of regular status meetings. Add new risks that evolve during the project, and remove risks that are no longer a threat.

Test Automation: The SQE group can be invaluable in automating the test process. They may write shell scripts or implement a more sophisticated test harness system, or utilize a number of open source and commercial test automation solutions. Test automation can apply to regression testing, functional testing, performance testing, security testing, and by using virtual machine and hypervisor technologies, platform and installation testing. The risk level of the product is an indicator of the level of testing required. The higher level of risk, the more depth and types of testing are needed. As mentioned in the stage three under static analysis, the SQE group may also fix the code based on static analysis results. One pattern that emerged from static analysis was a questionable coding practice that is repeated thousands of time in the code, for instance, using an input or string manipulation command that depends on a delimiter for determining string length rather than depending on the string itself to do it. Many languages contain alternate source code statements that would set an upper limit on the string length so an overflow condition could not be encountered. SQEs can write code fixing macros or special automated scripts that can replace all the instances of the offending statements with more secure code. Information learned from these experiences makes good input for the next iteration of the coding template.

Best Practice Forum: As development teams begin to see the value of Software Quality Engineering they start to share their experiences with their colleagues on other projects. Regular meetings, forums, or blogs to share what works and what does not work can become a regular part of the enterprise's culture. A safe environment must be created first, where it is okay to share failures. Management should be discouraged from participating in these forums, as the purpose is to improve process and to openly discuss ideas. Ideas for process improvement should flow from those who actually do the work. By participating in best practice forums, the teams learn what works and doesn't work for their colleagues in a safe, non-threatening environment. In many cases these discussions provide the justifications needed to lead a team to make changes to their processes and tools. The SQE group is an observer in these meetings and may provide technology details to aid in the developers understanding of terminology.

Benchmarking: Enterprise level best practice forums may lead SQEs and developers to be curious about other enterprises or even other industries, leading to the practice of benchmarking. Our SQE's have benchmarked other National Laboratories in the United States and the United Kingdom. The results of these benchmarks have resulted in the adoption of common test tool automation and software development estimation methods. SQEs have ventured out to unrelated industries such as a large California peanut roasting and packaging plant[13] and a world class custom motorcycle builder[14].

From the peanut plant the lesson of software security was re-emphasized. The plant had recently had one of its large and expensive peanut packaging machines stolen. This machine takes a very large truck to carry and certainly is not something that could be sold easily on eBay. The large machine was left outside temporarily as part of a refurbishing effort. The theft was most likely an inside job or done by someone in the packaging industry. Thieves will go to great lengths to steal things if there is a way to do it. As a result of this benchmark, all of our scientific codes are now run though static analyzers and detected weak security practices in the code are fixed. We had always believed that since our codes are not public facing, security was not a top priority.

From the motorcycle production line we learned about safety. Three mechanics assemble each motorcycle. One mechanic will take the new bike on its first test ride, but which of the mechanics will take the first ride is unknown during assembly. Needless to say the motorcycle mechanics are highly motivated to tighten down all the bolts and assemble things correctly. Also we learned that on these high end bikes, all the wiring is run through the frames. Not only does this eliminate unsightly cable bundles, it protects the wires. Most of the accessories (lights, mirrors, foot pegs) were derived from separately sold bolt-on accessory parts. What does this have to do with software development? How about value of pairs programming and involving real users and stakeholders in the project all along, data hiding and encapsulation as a good design practice, and the concept of using "plug ins" to enhance IDE's like Eclipse. A stage four SQE should always be searching for ways to improve.

Phrases that may be heard during Stage Four are:

Developer	Wow, I thought having lots of tests was required
Manager	Geez, I'm getting less complaints
SQE	Code coverage shows these few tests are most important
Stakeholder	How does this compare to others in this industry?

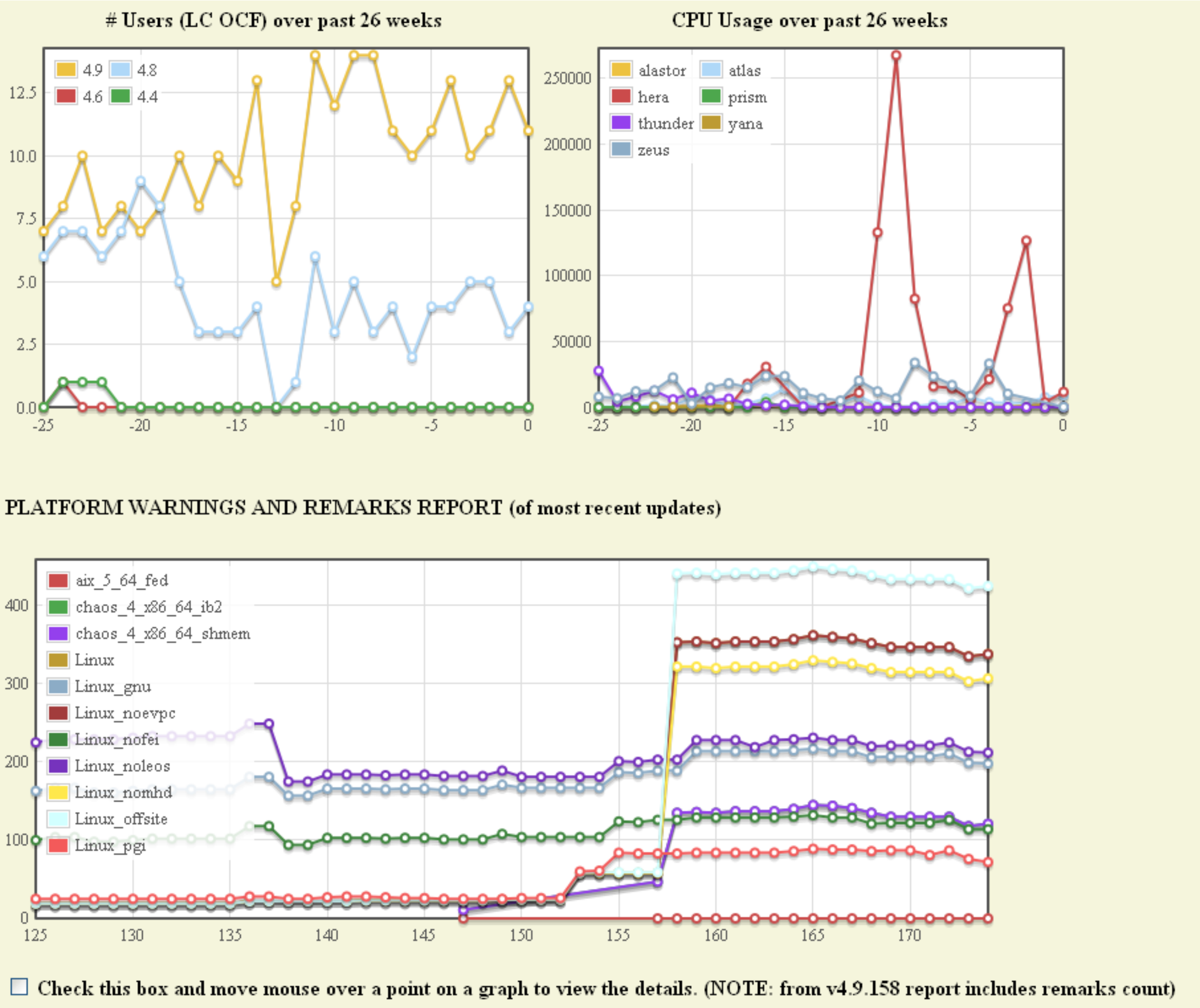
Some of the signs of a project moving from Stage Four to Stage Five include:

- Developers asking for more effective test cases
- Regular use of tools to document processes and decisions

Stage Five: Implement Improvements

At this stage, the development team is asking for more help with automating anything and everything to help simplify their lives. They realize some of the tasks that are taking too much of their time, such as constantly fixing their build system for each different platform, which makes the build system quite brittle. The SQE group can automate the build efforts by using tools, such as Autoconf, Ant and Electric Cloud, to develop scripts that reliably find the needed files and libraries on most platforms. It is important that the developers be able to override any defaults in these scripts as their knowledge of their code and computing platform leads to more efficiency.

Trending: While the appeal of the dashboard report with gauges may seem initially appealing, developing reporting tools to keep track of trends over time of measured information is useful for determining feedback from process changes or new tools. Sometimes the trending information can be quite simple, such as the level of compiler warnings. Complier warnings can be a useful source of important information about potential problems in the code. All too often these warnings get ignored, because under pressure the developers are just looking to get an error free compile. Keeping track of the number of compiler warnings can be used as a trigger to stop and fix up the code when they exceed a certain threshold. The information to capture and display compiler warnings can be scooped up using scripting languages like Perl or Python and displayed on a project web page. The bottom graph in Table 12 was drawn using a Python based Goggle plotting package. The warning levels are plotted for various platform types. The big jump in warnings on build 156 got the SQE's attention. Investigation showed that the compiler warning level has been turned up for that and subsequent builds. Other graphs shown are number of users and CPU hours consumed over time. Building trending tools is a prime activity of stage five SQE organizations.



PLATFORM WARNINGS AND REMARKS REPORT (of most recent updates)

☐ Check this box and move mouse over a point on a graph to view the details. (NOTE: from v4.9.158 report includes remarks count)

Table 12. Trending

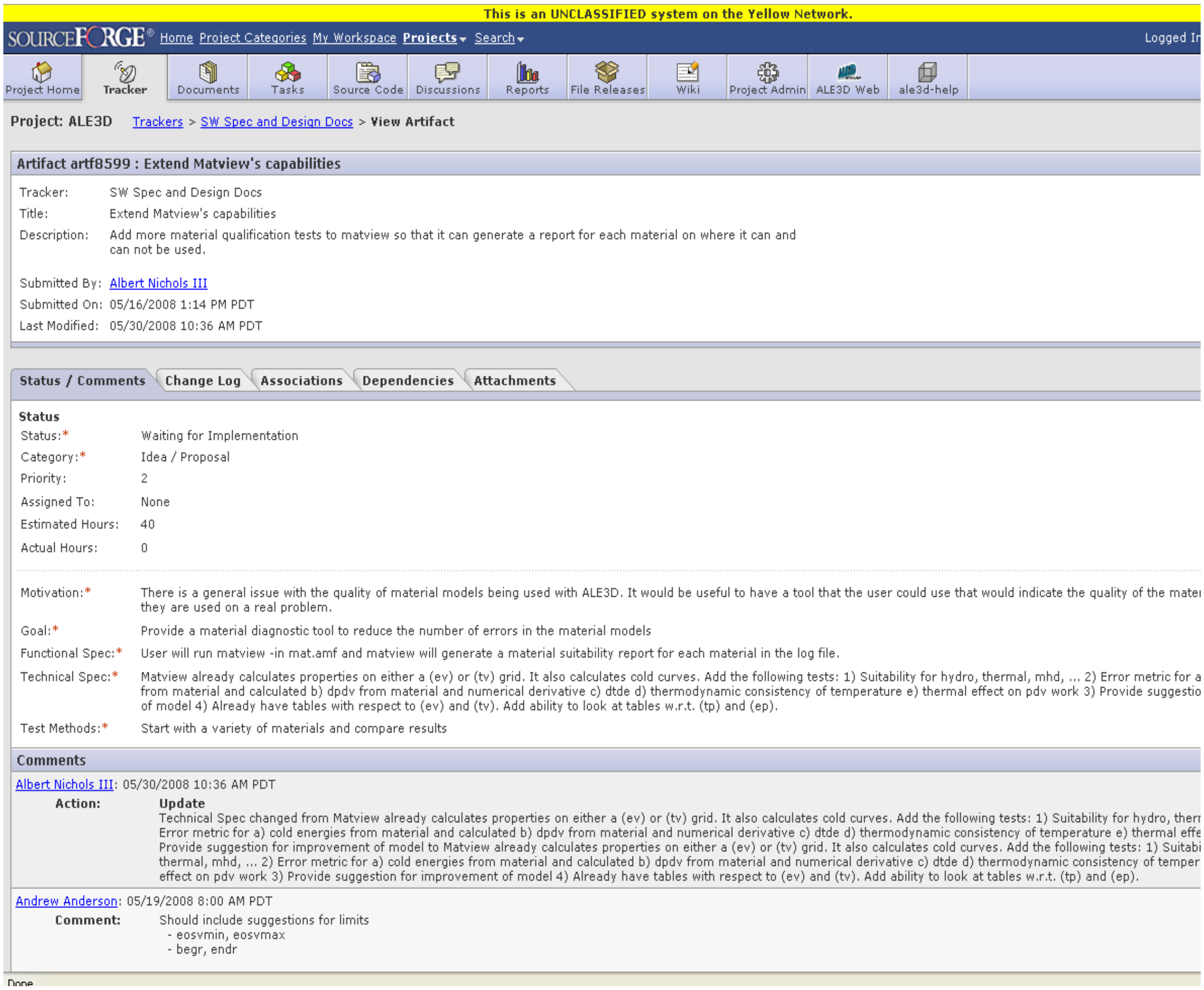
Test Improvement: Test case code coverage is conducted with regular frequency either by the SQE or developers. These results are used to continuously improve the test case suites and identify those tests that are best for any given feature. In many cases, the group will develop web-sites and email messages to report test results to the development team. Defect trends and metrics are developed from this information that can assist in identifying systemic issues that need to be addressed to improve the quality of the code and development processes. Automated test suites should be reviewed on a regular basis to assure the tests are still relevant and passing (or failing) for the right reasons. One developer group added an arbitration script, if the regression tests start failing, what was the last source code change made before the failures started?

Pilots: Piloting is the SQE practice of trying out new tools and techniques in a "sandbox" environment where success or failure during evaluation will not impact the productivity of the developers. The SQE at this stage of maturity should be involved in how to automate all phases of the software development lifecycle. Their goal is to simplify the developers' lives so that defects are prevented from ever getting into the code. SQE's spend time investigating and piloting new technologies, suggesting to the development team and management those that hold the most promise of being appropriate for the team.

It is important to realize that many times a suggestion from the SQE group may not be adopted right away without a proof of concept first to demonstrate the advantages of the new tool or technique. The development team may not have the time to learn a new tool or may not see how it helps them without actually using the tool. However, even if initially rejected by developers, the SQE's time has not been wasted as it set the groundwork for the time when the development team realizes that indeed, the new capability does simplify their lives.

Tool Usage: By this stage the development teams have many tools in place that they are using as part of their 'way of doing business'. The SQE group needs to monitor how the teams are using the tools in order to encourage efficient and appropriate use. At times, the development team may partially use a tool but not follow through with complete use. As an example, a project is using an issue tracking tool. They are diligent in entering, reviewing and responding to entries. However, they are lax to actually close out issues within the tool. Having issue trackers, for instance, is a two edged sword. On the one hand it assures that discovered defects will not be lost or fall through the cracks, on the other hand, aging reports from the tools will show management or auditors if defect fixing is being neglected in the pursuit of more glamorous tasks.

Pre- Design Capture: One of the last areas to mature in our software engineering environment was in design. With today's high level languages and visual tools the temptation to go from requirement description to code is high. While initially there may not seem anything wrong with the idea of not capturing design artifacts, design trade off discussions or reasons for choosing one design or architecture over another may be useful in the future. To accommodate this notion, the SQE's worked with the developers to develop a design template that is part of any design discussions. The template can be filled out during meetings electronically or updated by participants. The idea is to capture why approaches are chosen. For instance choosing MPI over multi-threading, a binary sort over a quick sort, or why the data is organized the way it is. Forums, wikis, or blogs can serve the same purpose. The idea is to grab the information in real time as part of the process. Table 13a shows a page from a pre-design discussion using the Source Forge Enterprise Edition tracker tool with an SQE crafted template. The pre-design information is version controlled, collaborative, and searchable so that it can be retrieved in the future.



Project: ALE3D Trackers > SW Spec and Design Docs > View Artifact

Artifact artf8599 : Extend Matview's capabilities

Tracker:	SW Spec and Design Docs
Title:	Extend Matview's capabilities
Description:	Add more material qualification tests to matview so that it can generate a report for each material on where it can and can not be used.
Submitted By:	Albert Nichols III
Submitted On:	05/16/2008 1:14 PM PDT
Last Modified:	05/30/2008 10:36 AM PDT

Status

Status:*	Waiting for Implementation
Category:*	Idea / Proposal
Priority:	2
Assigned To:	None
Estimated Hours:	40
Actual Hours:	0
Motivation:*	There is a general issue with the quality of material models being used with ALE3D. It would be useful to have a tool that the user could use that would indicate the quality of the mater they are used on a real problem.
Goal:*	Provide a material diagnostic tool to reduce the number of errors in the material models
Functional Spec:*	User will run matview -in mat.amf and matview will generate a material suitability report for each material in the log file.
Technical Spec:*	Matview already calculates properties on either a (ev) or (tv) grid. It also calculates cold curves. Add the following tests: 1) Suitability for hydro, thermal, mhd, ... 2) Error metric for a from material and calculated b) dpdv from material and numerical derivative c) dtde d) thermodynamic consistency of temperature e) thermal effect on pdv work 3) Provide suggestio of model 4) Already have tables with respect to (ev) and (tv). Add ability to look at tables w.r.t. (tp) and (ep).
Test Methods:*	Start with a variety of materials and compare results

Comments

Albert Nichols III: 05/30/2008 10:36 AM PDT

Action: **Update**

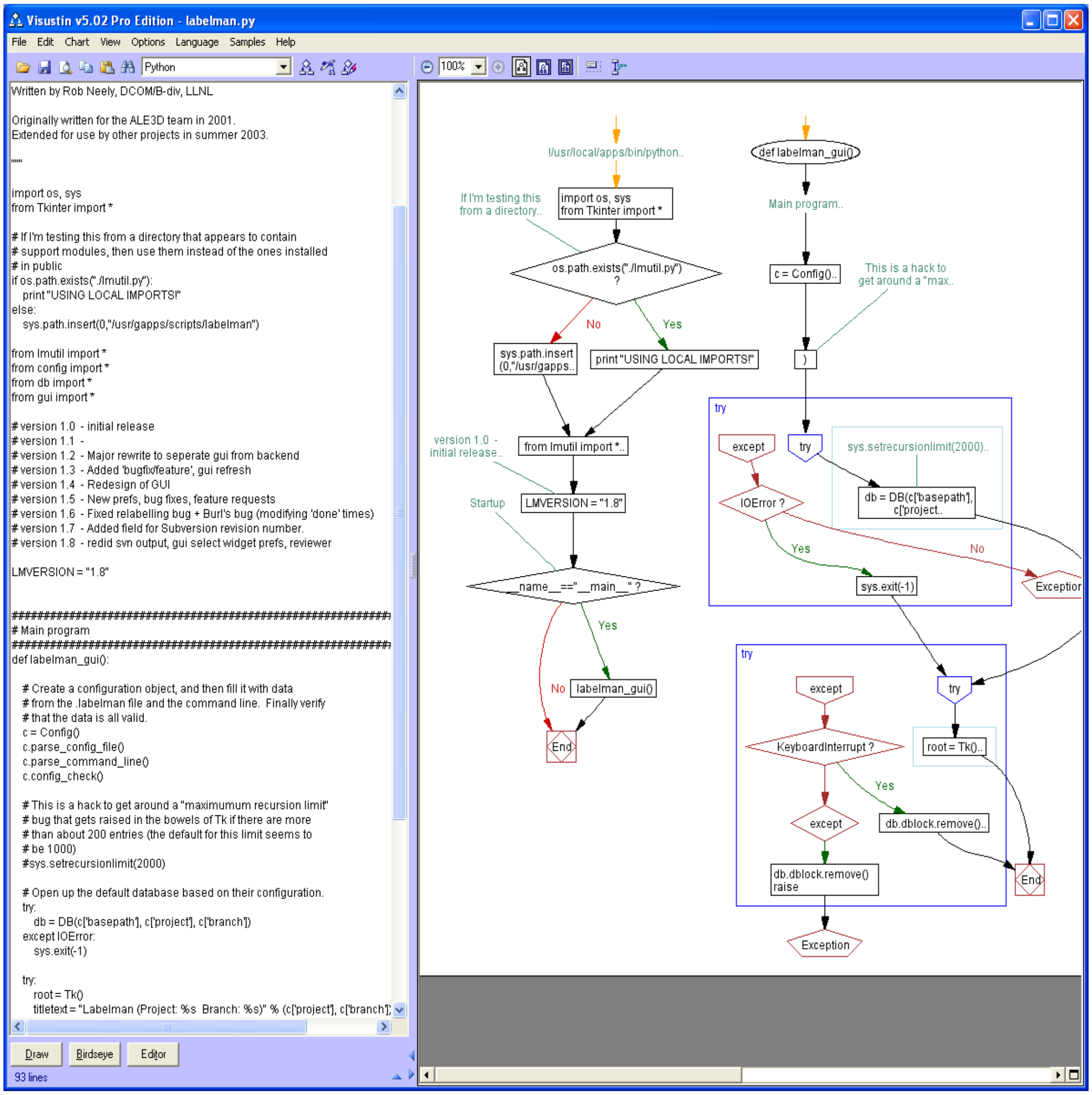
Technical Spec changed from Matview already calculates properties on either a (ev) or (tv) grid. It also calculates cold curves. Add the following tests: 1) Suitability for hydro, therr
Error metric for a) cold energies from material and calculated b) dpdv from material and numerical derivative c) dtde d) thermodynamic consistency of temperature e) thermal effe
Provide suggestion for improvement of model to Matview already calculates properties on either a (ev) or (tv) grid. It also calculates cold curves. Add the following tests: 1) Suitabi
thermal, mhd, ... 2) Error metric for a) cold energies from material and calculated b) dpdv from material and numerical derivative c) dtde d) thermodynamic consistency of temper
effect on pdv work 3) Provide suggestion for improvement of model 4) Already have tables with respect to (ev) and (tv). Add ability to look at tables w.r.t. (tp) and (ep).

Andrew Anderson: 05/19/2008 8:00 AM PDT

Comment: Should include suggestions for limits
- eosvmin, eosvmax
- begr, endr

Table 13a. Pre-Design Capture

Post Design Capture: What about legacy software that has already been written? What benefit can be achieved by going back and documenting the design? Probably not much and most likely no one is going to want to do it. To solve this long standing dilemma the SQE's found a flow charting tool (Visustin, from Finland) which works with most all contemporary and legacy languages. The legacy code is quickly flow charted and made available under the Source Forge document repository for those who will be maintaining or modifying the code in the future. Two other tools are also used to analyze existing source code from a design perspective, those are Klocwork, which identifies paths between files and modules and cyclic relationships, and McCabe Battlemap which can show the calling hierarchy. Both tools provide code and object oriented metrics which can be useful for prioritizing testing and refactoring. Table 13b shows an example Visustin flow chart of some Python code. The tool allows output in common graphical formats as well as direct export to MS Word, MS Power Point, and MS Visio.


Visustin v5.02 Pro Edition - labelman.py
File Edit Chart View Options Language Samples Help
Python
100%
Draw
Birdseye
Editor
93 lines
!/usr/local/apps/bin/python..
If I'm testing this from a directory..
import os, sys
from Tkinter import *
os.path.exists("./lmutil.py") ?
No
Yes
sys.path.insert (0,"/usr/gapps..
print "USING LOCAL IMPORTS!"
version 1.0 - initial release..
from lmutil import *..
Startup
LMVERSION = "1.8"
__name__ == "__main__" ?
Yes
No
labelman_gui()
End
def labelman_gui()
Main program..
c = Config()..
This is a hack to get around a "max..
)
try
except
try
sys.setrecursionlimit(2000)..
IOError ?
db = DB(c['basepath'], c['project..
Yes
No
sys.exit(-1)
Exception
try
except
try
KeyboardInterrupt ?
root = Tk()..
Yes
except
db.dblock.remove()..
db.dblock.remove() raise
End
Exception

```
Written by Rob Neely, DCOM/B-div, LLNL

Originally written for the ALE3D team in 2001.
Extended for use by other projects in summer 2003.

"""

import os, sys
from Tkinter import *

# If I'm testing this from a directory that appears to contain
# support modules, then use them instead of the ones installed
# in public
if os.path.exists("./lmutil.py"):
   print "USING LOCAL IMPORTS!"
else:
   sys.path.insert(0,"/usr/gapps/scripts/labelman")

from lmutil import *
from config import *
from db import *
from gui import *

# version 1.0  - initial release
# version 1.1  -
# version 1.2  - Major rewrite to seperate gui from backend
# version 1.3  - Added 'bugfix/feature', gui refresh
# version 1.4  - Redesign of GUI
# version 1.5  - New prefs, bug fixes, feature requests
# version 1.6  - Fixed relabelling bug + Burl's bug (modifying 'done' times)
# version 1.7  - Added field for Subversion revision number.
# version 1.8  - redid svn output, gui select widget prefs, reviewer

LMVERSION = "1.8"


##############################################################
# Main program
##############################################################
def labelman_gui():

   # Create a configuration object, and then fill it with data
   # from the .labelman file and the command line.  Finally verify
   # that the data is all valid.
   c = Config()
   c.parse_config_file()
   c.parse_command_line()
   c.config_check()

   # This is a hack to get around a "maximumum recursion limit"
   # bug that gets raised in the bowels of Tk if there are more
   # than about 200 entries (the default for this limit seems to
   # be 1000)
   #sys.setrecursionlimit(2000)

   # Open up the default database based on their configuration.
   try:
      db = DB(c['basepath'], c['project'], c['branch'])
   except IOError:
      sys.exit(-1)

   try:
      root = Tk()
      titletext = "Labelman (Project: %s  Branch: %s)" % (c['project'], c['branch'];
```

Table 13b. Post Design Capture

Independent Audits and Assessments: One of the SQE concerns, at stage five maturity level, is does the project or enterprise really see itself accurately? Have we become so familiar with the development process that we can no longer see shortcomings or be objective? To mitigate this risk the SQE does independent audits. Unlike the smaller internal audlets, these independent audits are conducted by teams which consist of SQEs from other parts of the enterprise and from other enterprises. They are costly to do, so they are done about every three years, but have been most helpful in finding out what we don't know that we don't know. Experience has taught us to use auditors (or assessors) who are experienced in our industry, and do not have a hidden agenda, such as laying the groundwork for obtaining a subsequent consulting contract to fix our shortcomings. From these independent audits SQEs have learned to schedule internal audlets so they do not get pushed aside and to schedule reviews of automated test suites to assure they get periodically reviewed. The independent audits and assessments have also pointed out our noteworthy practices and given some new ideas for the auditors to take back to their enterprises.

Phrases that may be heard during Stage Five are:

Developer	I want the SQE on my development team
Manager	The SQE is my 'go to' person
SQE	How much more do you want me to do?????
Stakeholder	Wow, this software works correctly right out of the box!

What's Next

What happens after Stage Five? Well you may have guessed the answer. In order to achieve stage five SQE performance each of the SQEs has polished their developer skills to a high degree. Because of this, many of the SQEs are encouraged to join the developer ranks. In Stage Five the development team wants the SQE as a member of their team. The developers recognize that the SQE brings skills to the team, understands their challenges, and sees the 'big picture' and ultimately, the developers trust the SQE to modify their code and write new code. As these SQEs go off to developer tasks, they are replaced with more junior staff, meaning the maturity level after stage five may return to earlier stages. This is because the new SQE's will think they know it all; start whining about developers, find documentation to their liking, etc. etc. This process is similar to the phases of team development as described by Dr. Bruce Tuckman's model of forming, storming, norming and performing. In Dr. Tuckman's model, the first three stages are when the team members get to know and trust each other, when that happens they perform well. The model described here has the same type of phases with a more diverse team.

Summary

This white paper takes the perspective that Software Quality Engineering moves through stages of maturity just like development does. It has attempted to define the relevant terminology and define the purpose of SQE. This paper suggested empirically derived traits of each SQE maturity stage. Also, that Software Quality Engineering is about more than testing. Because real people are involved in software development and Software Quality Engineering not all the traits may be true in each stage in your enterprise. In any case, our hope is that this white paper may provide you with some ideas and hope for the future in your quest to pursue improved Software Quality Engineering.

[1] Pressman, Roger. *Software Engineering: A Practitioner's Approach.* Sixth Edition, International, p 746. McGraw-Hill Education 2005.

[2] DeMarco, T., "Management Can Make Quality (Im)possible," Cutter IT Summit, Boston, April 1999

[3] McConnell, Steve. *Code Complete* First Ed, p. 558. Microsoft Press 1993

[4] Gerald M. Weinberg (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold, pg 146-147

[5] Pope, Gregory, "Why Software Quality Assurance Practices Become Evil!", http://www.stickyminds.com/s.asp?F=S8375_ART_2

[6] Pope, Gregory, "Why Software Quality Assurance Practices Become Evil!", http://www.stickyminds.com/s.asp?F=S8375_ART_2

[7] Collins, Jim, *Good to Great,* First Edition, HarperCollins Publisher, Inc., 2001.

[8] Boehm, Barry, Chris Abts, A. Windsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, Bert Steece – *Software Cost Estimation with COCOMO II,* Prentice Hall, Upper Saddle River, N.J. 2000

[9] Pressman, Roger. *Software Engineering: A Practitioner's Approach*. Sixth Edition, International, p 747. McGraw-Hill Education 2005.

[10] Boehm, Barry. *Software Engineering Economics*, Prentice-Hall, 1981.

[11] Su, Qiang, Shi, Jing-Hua, Lai, Sheng-Jie, "The Power of Balance", Quality Progress, February 2009, pp. 32-37.

[12] Kerzner, Harold, *Project Management – A Systems Approach to Planning, Scheduling and Controlling*, 6th Edition, Van Nostrand Reinhold, New York, NY, 1998.

[13] Nature Kist snacks, Stockton California, Ron Mozingo owner.

[14] Arlen Ness Motorcycles, Dublin, California, Arlen Ness owner.